コンピュータ・サイエンス研究書シリーズ ⑧

システム・プログラム II

マサチューセッツ工科大学教授 ドノバン著
筑波大学電子・情報工学系教授 池田克夫訳

日本コンピュータ協会

コンピュータ・サイエンス研究書シリーズ ⑧

システム・プログラム II

John J. Donovan著
京都大学情報工学科助教授　池田克夫訳

日本コンピュータ協会

systems programming

訳 者 序 文

本書は**システム・プログラム**と**オペレーティング・システム**に関して，“それが何であるか（what）”，“それをどのようにして使用すればよいか（how）”だけではなく，“それは何故に，どのような原理に基づいているのか（why）”を統一的な方法論によって述べている．ソフトウェア工学の教科書があまたある中で統一的な考え方に立ってその原理を述べた書物はきわめてまれである．本書は原著者の序文でも述べられているとおり，MIT における計算機の科目の教材として使用されてきたものを出版したものであるが，さすがは世界の最高水準といわれるだけあって，その教育内容は実に充実していることが感じとられる．その方法は単に高度のことを教えるというより，高度のことまで通用する物の考え方を深く掘り下げてみっちり仕込んでいるといえよう．

本書はアセンブラから始め，マクロ・プロセッサ，ローダ，コンパイラと進み，オペレーティング・システムの資源管理を述べているが，最初のアセンブラの章で示された言語処理プログラムやシステム・プログラムの基本的な原理はマクロ・プロセッサ以降の章でも統一的に使用されているし，コンパイラのコード生成段階はマクロ・プロセッサそのものであるというくだりになると今まで難しそうに見えたコンパイラの構造がいともあざやかに浮び上ってくるのである．オペレーティング・システムについても資源管理という面から統一的に論じている．また，第9章で述べられているセグメント化された番地空間における情報の管理は，今後の計算機システムにおいて本質的に重要な役割を果たすであろう．

各章末に付けられた演習問題は本文の議論を一段と発展させ，考え方を整理する上で有用である．原著では解答は別冊となっているが，本書では読者の便を考えて第Ⅱ巻の巻末に解答を付けておいた．本書を学ぼうとする読者は第1章から順を追って問題をも解きながら進まれるのがよい．

本書は，序文の中でも述べられているとおり，計算機工学を専門としようと

する人達にとって計算機およびシステム・プログラムの概念と設計を修得させるのに最適であるし，計算機を専門としている人達にとっては考え方を整理するうえで大変役に立つであろう．また，計算機を専門としない人達にとってもきわめて自然な形で計算機を理解させることができるであろう．

本書を読むに際しては計算機の構造とプログラミングについての知識が要求されている．本書は計算機の構造やプログラミングのための入門書ではないから，これらの事項を最初から学ぼうとする人々は他の参考書で勉強してから本書を読んだ方が良い．例えば PL/I を紹介している第6章にしても，通常の高水準言語を知っているとした上で，上級の機能の意義を述べている．本書が教科書として使用されるときは先生が適宜補足されるであろうからこれらの点に関する問題はないであろう．

本書を翻訳するに当り原文中で明らかに間違っていると思われる個所は訳者の責任において訂正しておいた．単に翻訳したでけでは意味の通じない個所は適宜文章を補っておいた．

訳者が本書の発刊を知ったのは MIT に客員として留学していたときであった．早速買い求めてみたところ上述のとおり優れた内容であることを見出した．訳者自身が京都大学でシステム・プログラムの講義を担当していることもあり教科書として最適であると考えたので翻訳を志したのであるが，東京大学助教授石田晴久氏のお取り計いでこの計画が実現できた．

本書を翻訳出版するに当って，京都大学大学院生杉山　守，小野隆夫，岡島一郎，町田哲夫の諸君は翻訳原稿に関して有意義な意見を述べられた．原稿の浄書と校正を愚妻和子が，原稿の整理を中谷いつ子君が手伝ってくれた．

これらの方々の協力に深く感謝する次第である．

1974年8月

池田克夫

序　　文

目　　的

本書では，アセンブラ，マクロ，ローダ，コンパイラおよびオペレーティング・システムの使用とインプリメンテーションを含め，システム・プログラム全般にわたって述べようとしている．すなわち，新しい計算機用言語と新しいオペレーティング・システムに関連して，その構成要素を詳細に示し，その設計の問題点を明らかにする．これらの問題点としては構文や意味論といった従来のコンパイラの問題に加えて，記憶装置割当てや競合の生じる参照の問題がある．また今日では，ファイル・システム，多重処理，多重プログラミングなどがオペレーティング・システムにおいて普通のこととなっている．計算機科学のより公式的な面を紹介するために，形式的体系による表現とプログラム言語への応用をも含めた．

本書は教科書として，演習問題に特に重点を置いて書いてある．読者は高水準言語での経験を持っているものと仮定している．

本書の内容は可能な限り機械と独立にするよう努力しており，実際に，幾種かの違った型の計算機を用いた教育に使用されている．しかしながら，真実味を増すために IBM 360/370 型の計算機を具体例とし，　またコンパイラの議論には PL/I に示される特徴を持った言語を例として用いている．

本書は ACM（Association for Computing Machinery）の計算機科学におけるカリキュラム委員会[1]　が提示したカリキュラム 68 の 6 つの科目の内容を網羅している．基礎科目である計算機の構成とプログラミング（Ｂ２）の内容は第 1 章から第 5 章に，プログラム言語（12）とコンパイラの構成（15）の内容は第 6 章，第 7 章および第 8 章に，システム・プログラム(14)，進んだ計算機構成(Ａ２)，および，大規模情報システム（Ａ８）の内容は第 9 章において扱っている．

1　CACM vol. 11, no. 3, p. 151 (March 1968) 参照.

序　　文

本書の使用法

本書は主として次の3つの使用法がある．(1) 1～2学期分のシステム・プログラムの学部向き科目の教科書として，(2)専門家用として，(3)大学院学生の参考書として．

具体的には本書は次の科目の要求にかなうものとして使用されている：

1. （例えば FORTRAN や PL/I のプログラム講習的な科目に引続く）計算機科学の学部教育の最初の科目
2. 計算機関係以外の一般教育科目
3. ソフトウェアの上級科目
4. 実際的な問題に重点を置くソフトウェア工学の科目
5. 計算機科学の大学院生に対する概論または導入的科目

MIT (Massachusetts Institute of Technology：マサチューセッツ工科大学）において，本書はディジタル計算機プログラミング・システム 6.251 という学部向き科目で使用されている．この科目には伝統があり，学期毎に 350 人が選択し，最も学生数の多い選択科目の1つである．また，この科目は最もやりがいのある科目の1つでもある．MIT でこの科目は上述の目的により開設されている．

また，同じ教材は2学期間の大学院科目にも使用されている．最初の学期では計算機の構成，アセンブラ，マクロ，ローダ，I/O プログラミングおよびオペレーティング・システムの話題が取り扱われる（第1，第2，第3，第4，第5章，および第9章の一部分）．次の学期では プログラム言語，コンパイラの設計，形式的体系，および最初の学期で扱われなかったオペレーティング・システムの他の機能が論じられる（第6，第7，第8，および第9章）．

同じ科目は Honeywell，米国水底システム・センター (U. S. Underwater System Center)，Martin Marietta，その他の幾つかの企業で，主としてシステムの構成要素の設計問題に焦点を合わせ，形式的体系（第7章）を省いて教

えられている.

この教科書はフランスの SESA の上級プログラムの科目や, Boston の Lowell School の2年間の技術者養成課程の科目でも使用されていた.

Lubbock, Texas の Texas 工科大学でのプログラムの科目は, 最初 MIT の講義のビデオ・テープを用いて行なわれた. MIT のビデオ・テープは次の週に Texas に送られて再生された. このようにして授業を他で行なうことは, Texas の学生と MIT の学生に試験で同じ問題を出題し, 成績に目立った差がなかったので, 有効であることが実証された.

MIT でこの科目に関連して使用された機械は IBM 360/370 型計算機である. 米国海底音響研究所 (U. S. Underwater Sound Laboratories) では UNIVAC 1108 が使用されていた. Honeywell では各種の Honeywell の機械が使用されていた.

学生がアセンブラ言語や PL/I の経験を持っていることは内容を理解する上で役に立つ. しかしこれらの経験を持っていない場合には, 計算機のマニュアル, 講義, および本書の第2章と第6章をアセンブラ言語や PL/I 入門の手引きとして使用することができる.

専門家または大学院学生が使用するとき, 本書には IBM 370 と PL/I に関して十分な内容が含まれており, 残りの部分を読むために他の本を参照しなくてもよい.

アセンブラやマクロなど従来のシステム要素の議論に加えて, 本書は現在発行されている教科書にはまだあまり述べられていないシステム・プログラムの重要な特徴, コンパイラや記憶装置割付け, 回帰呼出し, オペレーティング・システムおよび I/O プログラミングなどの高度の問題に特に重点を置いている.

謝　辞

本書自身がこれに寄与した多くの人達の尽力と技術的能力に対する謝辞である．ここでは極く一部分の人々の名前だけしかあげることができない．

Stuart Madnick 氏の貢献は特筆すべきものである，Madnick 氏は私の知っている限り最も優れたシステム・プログラマである．氏は私と一緒に5年間，助手および講師として，本書を使用する MIT の科目（ディジタル計算機プログラミング・システム）の講義を行なった．氏はファイル・システムとオペレーティング・システム概論の部分を受持った．また氏はすべての章を読みその内容に寄与した．私は氏の技術的貢献と忠実さに感謝したい．

MIT のディジタル計算機プログラミング・システムはこれを担当した教授達によって発展してきた．これらの教授の名をここにあげておく

1959年	F. Verzuh 教授
1960年	F. J. Corbató 教授
1961年	W. Poduska および N. Haller 教授
1962年	J. Saltzer 教授
1963年	C. L. Liu, D. Kuck, F. J. Corbató および D. Thornhill 教授
1964年	J. Saltzer, D. Kuck, R. Fabry および T. Stokham 教授
1965年	W. Poduska, L. Hatfield, R. Baecker および R. M. Graham 教授
1966年	R. M. Graham 教授
1967年—現在	J. J. Donovan 教授

科目の内容はアセンブラから始まってローダ，探索法，デバッグ技法，最近になってオペレーティング・システムを含むに至ったので，個々の講師の貢献を厳密に述べるのは困難である．Corbató 教授は統一的な概念を設定し，事例

研究法を導入した．私の前任者 Graham 教授は私に多くの基本事項を教授されたとき，私の考え方に大きな影響を与えた．

過去5年の間，多くの助手の援助によりこの科目を IBM 360 に合せて再構成し，直接結合型のローディング方式，PL/I，高水準コンパイル技法，記憶装置とプロセッサの管理，および，最近ではI/O プログラミングを組み入れた．

毎年優れた助手のグループがこの科目のために働いてくれたので，350人もの学生を1学期に教えることができた．学生の広範な要求をさばいてくれたこれら助手各位の尽力と能力に負うところが大きい．過去数年間のこれら助手の長は Stuart Madnick, Steven Zilles, Richard Mulhern, William Michels, および Glen Brunk である．しかし他の多くの人々も様々な方法で寄与し，特に問題の設定に対して尽力してくれた．ここに幾人かの名をあげておく．David Quimby, Jerry Johnson, Michael Hammer, Benjamin Ashton, Orville Dodson, Steven Halfich, Norman Kohn, Martin Jack, Al Moulton, Chander Ramchandani, および Harris Berman. Muriel Webber 夫人は図版の作成をしてくれた．Max Byer は MIT の科目の管理に貢献した．

次の人々の多大な努力が本書の内容に貢献した．William Michels はコンパイラの章に大きく寄与した．Norman Kohn はマクロと形式的体系の章および参考文献の整理に寄与した．Sterling Eanes は形式的体系およびコンパイラの章を校正した．David Quimby は演習問題の編集を行ない，本書の草稿に意見を述べてくれた．Leonard Goodman は本書全部の校正を受持ってくれた．訂正された誤りに対しては氏に礼を述べたい．David M. Jones 氏は原稿を編集してくれた．Ellen Nangle はタイプを指揮し，原稿の準備をしてくれた．彼女はさらに，原稿，演習問題，解答および教師用マニュアルを編集した．

本書の中で報告されている研究の一部は国防省の上級研究計画局（Advanced Research Projects Agency）の後援で，海軍研究契約（Office of Naval Research Contract Nonr-4102(01))によって設立された研究プロジェクトである MIT のプロジェクト MAC において行なわれ，一部は MIT の電気工学科で行なわれた．

最後に，形式的にであると思われそうなのでためらっているのであるが，妻

謝　　辞

と５人の子供達に心から一言ありがとうをいいたい．私の仕事は，過去２年間，１日24時間，１週７日，そして，それ以前も長期間この原稿を用意するのにかかっていたから，家族の忍耐と援助に感謝している．

John J. Donovan

学 生 諸 君 に

本書はアセンブラ言語の深い知識を仮定してはいないが，この言語に経験を有する学生には理解しやすいであろう．第２章はアセンブラ語のプログラミングに関する基本的な事項を，第６章は高水準言語，特に， PL/I について述べている．アセンブラ言語および PL/I のプログラミングに関しては，他の参考書をも参照するのがよい．

第８章および第９章（コンパイラとオペレーティング・システム）は長く内容が盛りだくさんである．（これらの章だけで１つの科目ができるほどである.)これらの章は読者の都合の良いように幾つかの部分に分割してある．

工学の分野において，特に，計算機科学のように急速に発展している分野においては，関係の文献に目を通しておくことが必要である．第10章は参考文献を示している．活用してほしい．

本書はプログラミングを知っていることを前提としている．したがって，未経験の読者にとっては内容が高度で密度が大きく難しいだろうが失望しなくてもよい．

本書の内容は計算機科学の基本である．そしてよく勉強した学生は内容が適切で興味深いと思うだろう．この内容を勉強するためのより公式的な方法を私は知らない．

目　　次

第2部

第3部

第4部

第5部

第 8 章

コンパイラ

本章は次の2つの目的を持っている：

1. コンパイラの設計と研究の基本として使用できる一般的なコンパイラのモデルを示すこと．
2. 言語の組込みと特定の機能を使用する難しさおよび“経費”の評価を行なうこと．

この目的を果たすために本章を次のように3部に分けている：

第1部　簡単な例を示し，コンパイラの一般的なモデルを導入する．

第2部　モデルを吟味し，その働きを詳細に述べる．

第3部　このモデルを用いて第6章で論じられた高級な機能（たとえば，データ構造，回帰性，記憶域割付け，ブロック構造，条件およびポインタ）の組込みを示す．

第 1 部

8.1 問題の設定

コンパイラは入力として高水準言語で書かれたプログラムを受理し，出力として等価な機械語プログラムを生成する．第1部では簡単な PL/I プログラムを考え，それをコンパイルしようとするとき，直面しなければならない問題点を調べる．

図 8.1 の WCM と等価な機械語のプログラムを生成するためにコンパイラは何をしなければならないであろうか．それらは；

1. ある文字列が基本要素であると認識すること．たとえば，COST は変数，

```
WCM: PROCEDURE (RATE,START,FINISH);
        DECLARE (COST,RATE,START,FINISH) FIXED BINARY (31)STATIC;
        COST = RATE * (START-FINISH) + 2 * RATE * (START-FINISH - 100);
        RETURN (COST);
END;
```

図 8.1　MINI—PL/I プログラムの例

WCM はラベル，PROCEDURE は見出し語，および“＝”は1つの演算子であると認識する．

2. 要素の組合せが構文の単位であると認識し，その意味を解釈すること．たとえば，最初の文は3つの引数を持つ手続き名であり，次の文は4つの変数が31ビットの2進数であると定義し，3番目の文は7回の演算を必要とする代入文であり，最後の文は1つの引数を伴う復帰文であると認識する．
3. このプログラムのすべての変数に記憶域を割付け，番地を割当てる．
4. 適当な目的コードを生成する．

8.1.1 問題その1——基本要素の認識

原始プログラムを解析して適当な構文要素のクラスに分けることを**単語解析**(lexical analysis）という．プログラムは走査され，**図 8.2** に示したように分割される．この段階の操作では，単純な記号列処理の技法を使用している．原始プログラムは順番に走査される．基本要素，すなわちトークンは空白，演算子，および特殊記号により区切られ，記号，リテラル，または終端記号（演算子，見出し語）として認識される．

基本要素（記号とリテラル）は表に記入される．他の段階でその要素の用法と意味を認識すると，さらに情報（たとえば，精度，データの型，長さ，および記憶域のクラス）がこれらの表に記入される．

コンパイラの他の段階では各基本要素の属性を使用するので，この情報が参照できなければならない．各要素に関する情報は，そのまま他の段階に渡されることもあるし，原始記号列が“不偏記号”の列に変換されることもある．**不偏記号**は固定長で，構文要素のクラスを示し，また，基本要素に対応した表の欄へのポインタでもある．**図 8.3** はプログラマに使用される不偏記号を示している．

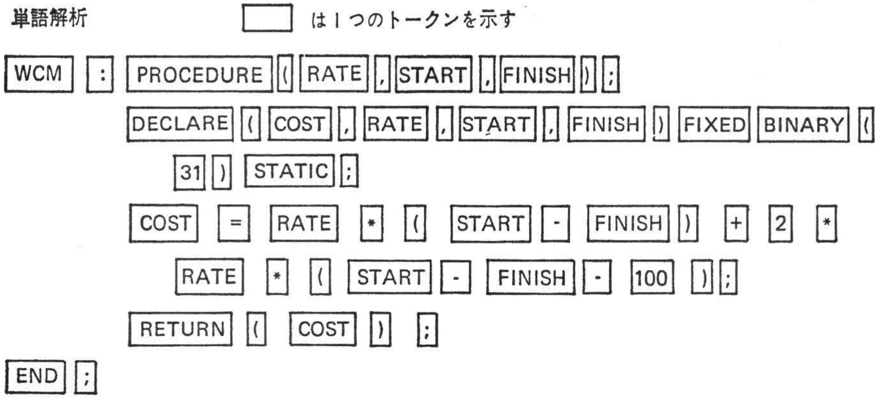


図 8.2 単語解析——例題プログラムのトークン

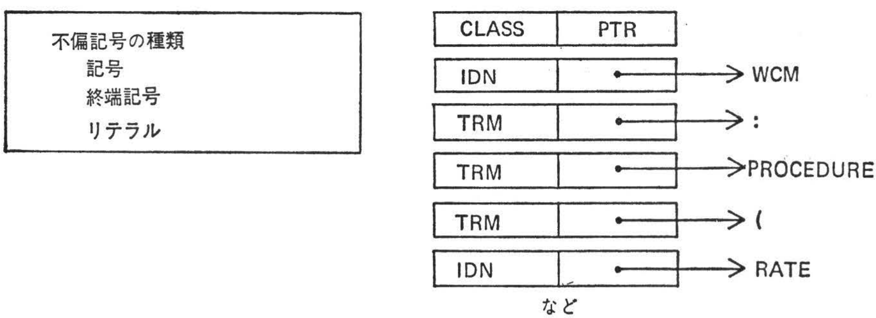


図 8.3 第1文の不偏記号

不偏記号は固定長であるので，これに変換しておくとコンパイラの後の段階がより単純になる．また，比較したいことは構文要素のクラスやポインタの値なのであって，長い記号列そのものではない．

単語解析の処理は1つのパスで行なうことができる．このときトークンの鎖や表から成るプログラムの中間形式が作られる．別の方式では必要とされるトークンのみを解析し，不要になれば直ちにそれを解放してトークン表の容量を小さくしている．現在のコンパイラはこれら両方の方式を用いている．どちらの単語解析法でも単語の誤り（たとえば，間違った字や不適当な記号）を発見し通知することができる．註釈文はプログラムの処理に無関係であるから，単

語（解析）段階で捨てられる．

なお，トークンの長さは1から31文字であるのに比べ，不偏記号の長さはこれより短く等長である．コンパイラの他の段階では主として不偏記号を処理するが，ポインタによってトークンの属性を参照することもできる．

8.1.2　問題その2——構文単位の認識と意味の解釈

プログラムがトークンや不偏記号に分解されると，コンパイラは次の2つのことを行なわねばならない：

1. 句（構文上の構造）を認識する．各句は意味を定めるものであり，意味を持ったトークンの列である．
2. 構造の意味を解釈すること．

これら2つのステップの第1は，原始プログラムの中の基本的な構文を認識し，分割することだけに関係する．この処理は**構文解析**（syntax analysis）として知られている．例では，文を句として考え，プログラムは**図8.4**のように走査され，分割される．

また，構文解析では構文の誤りを通知し，回復の手段を講じてコンパイラが他の誤りを続けて探すことができるようにする．コンパイラによってはプログラマが行なおうとしたことを推定し訂正を試みる．

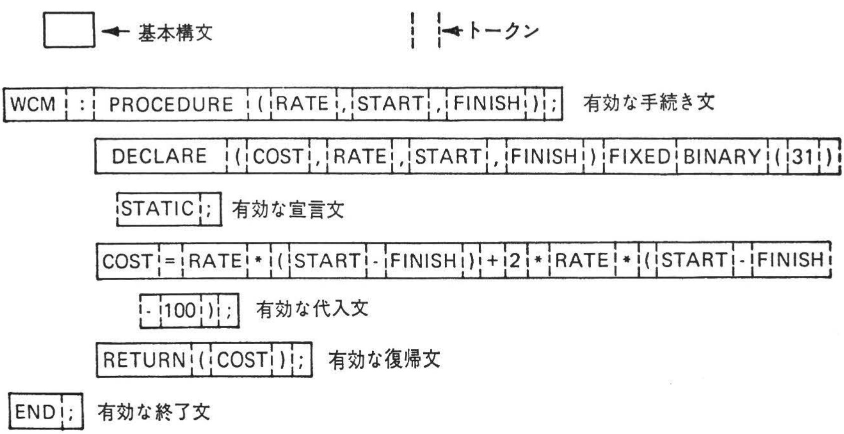


図 8.4　構文解析——例題プログラムの構文単位

文の構文が確認されると，第2ステップとして**意味**（semantics）が解釈される．各構文に対応して意味が定められている．これ（意味の表現のし方）は実際のコードの形の場合もあるし，中間形式の場合もある．

基本構造（構文解析の結果）を認識し，その意味を解釈する方法には多くの種類がある．第2部ではきわめて一般的な方法を選ぶ．それは原始言語の構文を定義する規則（**変形則** reduction）を使用する．これらの変形則は基本構文とその構造が認識されたとき実行される適当なコンパイラ・ルーチン（**活動ルーチン**）を定義する．その活動ルーチンは構造の意味を解釈し，コードまたは構造を示す中間形式を生成する．たとえば，変形則は，“記号に‘=’符号が続けば活動ルーチン‘ARITHMETIC-STM’を呼出せ”と指定する．

8.1.3 中間形式

構文上の構造が決定されると，コンパイラは各構造に対して目的コードを生成することができる．しかし普通には，コンパイラは原始プログラムの中間形式を作り出す．中間形式を用いることの利点は次の2つである：

1. 目的コードの最適化が可能となる．

2. **機械に独立な段階**（単語，構文，解釈）と**機械に関係する段階**（コード生成，アセンブル）とを分離することができる．

中間形式を用いるときに次の2つの問題が生じる：

1. 形式はどうするか．

2. 原始コードから中間形式へ変換する規則は何か．

中間形式は構文上の構造，たとえば，算術文，非算術文など，に関係する．

8.1.3.1 算 術 文

算術文の中間形式は**構文解析の樹**である．前章の形式的な手法，たとえば第7章の問1——一般的な構文の樹——によって**図 8.5** の構文解析の樹を得ることができる．算術文を構文解析の樹に変換する規則は次のようなものである：

1. 変数は樹の終端の節である．

2. すべての演算子に対して代数の規則によって示された順番で枝の2本ある樹（binary tree）を作り，左の枝には第1演算数を，右の枝には第2

演算数を対応させる.

例題のプログラムの算術文に対する構造の樹を図8.5に示す. この図は文の構造を見るのには具合が良いが, コンパイラにとっては実際的な方法ではない.

COST	=	RATE	*	(	START	-	FINISH	)	+	2	*	RATE	*	(	START	-	FINISH

-	100	)	;

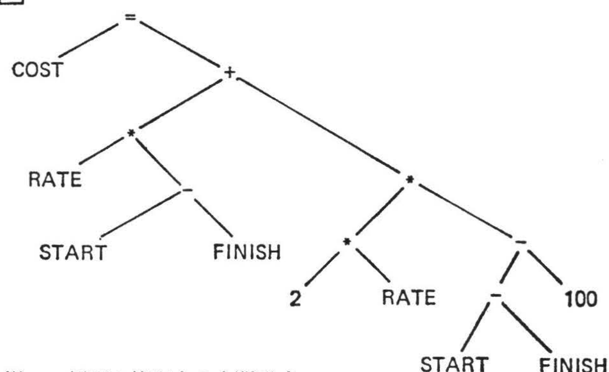


図 8.5 樹——例題の算術文の中間形式

コンパイラの中間形式としては**行列**と呼ばれる構文解析の樹の線型表現を用いることができる. 行列の中にはプログラムの演算が, 実行される順番に並べられている. 行列の各欄は, 1つの演算子と2つの演算数とを持っている.

COST	=	RATE	*	(	START	-	FINISH	)	+	2	*	RATE	*	(	START	-	FINISH

-	100	)	;

行列の欄番号	演算子	演算数1	演算数2
1	-	START	FINISH
2	*	RATE	M1
3	*	2	RATE
4	-	START	FINISH
5	-	M4	100
6	*	M3	M5
7	+	M2	M6
8	=	COST	M7

図 8.6 例題の算術文の行列

原始文		演算子	行列 演算数 1	演算数 2
RETURN (COST);	→	Return	COST	
END;	→	End		

図 8.7 RETURN 文および END 文に対する行列の例

演算数は，変数，リテラル，または他の行列の欄 M*i*（*i* は欄の番号を示す）を示す不偏記号である．前の算術文に対して生成される行列を**図 8.6** に示す．（理解を容易にするために欄には不偏記号を用いず実際の記号を用いている．）

樹（図 8.5）と行列（図 8.6）は算術文の等価な表現である．読者は，初等代数の規則を用いてこのような文の樹を作ることができる．コンパイラについて述べた本の多くは算術文から樹や行列を構成する処理を主題としている．本書ではこれを読者の演習問題として残しておくことにする（第 7 章問 1 を参照）．

8.1.3.2 非算術文

非算術文に対して正確な中間形式を作る問題は算術文の場合と同様である．DO, IF, GO TO などの文は行列の欄を並べるだけですべて置換えることができる（**図 8.7**）．行列の中の演算子は，適当なコードを生成するためにコンパイラの後の段階で定義される．

名 前	基 数	尺 度	精 度（ビット）	記憶域のクラス	番 地
COST	BINARY	FIXED	31	STATIC	0
RATE	BINARY	FIXED	31	STATIC	4
START	BINARY	FIXED	31	STATIC	8
FINISH	BINARY	FIXED	31	STATIC	12

図 8.8 記　　号　　表

8.1.3.3 非実行文

DECLARE や FORTRAN の DIMENSION といった非実行文は，コンパイラに変数の参照や割付けおよびそれに対応した記憶域を明らかにする情報を与える．これらの文に対する中間形式はないが，非実行文が持っていた情報は表の中に格納され，コンパイラの他の部分で使用される．例の DECLARE 文においては，変数 COST, RATE, START および FINISH の各々に対して，

解釈段階でデータの型，精度および記憶域のクラス(FIXED BINARY, 31 ビット，STATIC) を記号表の中に記入する (**図 8.8**)[1].

PL/I のような能力の大きい言語では，ある種の記憶域の割付けやデータの定義は実行時に行なわれるので，一部の表はコンパイラだけではなく目的プログラムによっても使用される．したがって，ある種の宣言の情報は実行時に使用するために目的プログラムへ渡される．

これまで，単語解析，構文解析および解釈（意味論）を論じてきた．次に，PL/I の例に関して，残る2つの問題がいかに解決されるかを述べよう．

8.1.4 問題その3——記憶域の割付け

時として，プログラムに要求される記憶域をあらかじめ適当にとっておかなければならないことがある．次の例の DCL 文は記憶域割付けのために適当な情報を与える．

DECLARE	(	COST	,	RATE	,	START	,	FINISH	)	FIXED	BINARY	(	31	)
STATIC	;													

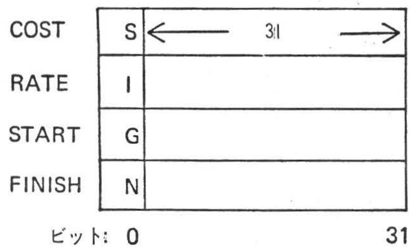
解釈段階は**図 8.8** に示す表の欄を構成する．

記憶域割付けルーチンは記号表を走査し，各単純変数に記憶場所を割当てる．32 ビットの固定小数点2進数の場合には最初のものに相対番地0，第2のものに4，……と割当てる．絶対番地はロード時までわからない．記憶域割付け段階では，これらの2進の 31 ビットと符号1ビットからなる4語はロード時に割付けられると仮定しなければならない．コンパイラは後の段階でこのように相対的に割付けられた番号を使用して適当に参照を行なう．

同様にして，行列の中間結果を格納する一時記憶用の場所（たとえば，M1, M2, M3, M4, M5, M6, M7）が割付けられる．

FORTRAN のようにすべての記憶域がコンパイル時に割付けられる言語に対しては，記憶域の割付けはやさしい処理である．表の中の欄を調べて，基

1 変数 RATE, START, FINISH に対して "STATIC" を宣言すると完全な PL/I コンパイラは誤りの警告を発する．PL/I はこれらを特別の記憶域クラスのパラメータとみなす．完全な PL/I でこのパラメータがどのように組込まれているかは付録第5節を参照してほしい．



数，尺度および精度を求め，適当な記憶域を確保する．しかし PL/I のような言語では STATIC 記憶域だけがこのようにして割付けられる．記憶域には他に 3 つの型があり（AUTOMATIC, CONTROLLED および BASED)，これらの割付けは上と異なっている．

自動，被制御または基底つきと宣言された変数に対しては，各変数に関する情報が目的デックに含まれていて，実行時に使用されて記憶域が割付けられる．

さらに，コンパイラは PROCEDURE, BEGIN および ALLOCATE などの文に対して，これらの変数に必要な記憶域を確保するために適当な目的コードを生成する．同様にして，END 文，RETURN 文および FREE 文は，この動的記憶域を解放する目的コードを生成しなければならない．すなわち，これらの文に対して行列の中に適当なコードが生成されるようにする欄を作らねばならない．

この型の記憶域を参照するのは難しいことである．通常コンパイラはこれが実際に割付けられるか否かについて知らないし，どこに割付けられるかについては見当もつかない．この型のデータの参照は，動的に割付けられたデータを指し実行時に内容を更新できるレジスタに対して相対的に行なうことができる．

8.1.5 問題その 4 ――コードの生成

コンパイラが，情報を行列および表に記入すると次に目的コードを生成することができる．

コード生成の 1 つの方法は，目的コードに関連する行列の処理の型を定義した表（コード生成表）を用意することである（図 8.9)．コード生成段階では行列を走査し，各欄に対して定義されているコードをその欄の演算数を用いて特

－, ＊, ＋, ＝に対する標準的なコード定義

```
－  L    1,&OPERAND1
    S    1,&OPERAND2
    ST   1,M&N
＊  L    1,&OPERAND1
    M    0,&OPERAND2
    ST   1,M&N
＋  L    1,&OPERAND1
    A    1,&OPERAND2
    ST   1,M&N
＝  L    1,&OPERAND1
    ST   1,&OPERAND2
```

ただし　& OPERAND 1 は行列の第1演算数
　　　　& OPERAND 2 は第2演算数
　　　　& N は判定される行列の欄番号である

図 8.9　コードの定義―― －, ＊, ＋, ＝に対する生成則

別化し，生成する．**図 8.9** のコード定義により行なったコード生成の処理を**図 8.10**に示す．この方法は，行列の中の演算子をマクロの呼出し，演算数をマクロの引数，生成表をマクロの定義として考えることもできる．

図 8.10 を注意深く見ると3つの疑問が生じる：

1. 行列から直接コードを生成するのは良い方法か．行列の行1および4は同一であり無駄なコードを作り出している．
2. 機械を最も有効に使用しているか．生成されたコードの行 12（ST 1, M4）および行 13（L 1 , M4）は無駄である．これらは計算結果に何等影響を与えない．また，IBM 360 の 16 個のレジスタの内2個のレジスタしか使用していないし，5つの型の命令の内1つの型（RX）だけしか使用していない．より高速の RR 命令は使用されていない．
3. 直接機械語を生成できるか．上の例ではアセンブラ語が使用された．行列の演算数の1つがラベルであり，そのラベルを持つ文が生成されていないときには何が起こるか．

これらの疑問の始めの2つは最適化の問題である．第1は中間形式（機械に独立の形）としての行列の最適性であり，第2は実際の機械コード（機械に関係する形）の最適性である．コンパイラは第2の問題よりも先に第1の問題を

行列				生成されたコード	
				L	1,START
1	−	START	FINISH	S	1,FINISH
				ST	1,M1
				L	1,RATE
2	*	RATE	M1	M	0,M1
				ST	1,M2
				L	1,=F'2'
3	*	2	RATE	M	0,RATE
				ST	1,M3
				L	1,START
4	−	START	FINISH	S	1,FINISH
				ST	1,M4
				L	1,M4
5	−	M4	100	S	1,=F'100'
				ST	1,M5
				L	1,M3
6	*	M3	M5	M	0,M5
				ST	1,M6
				L	1,M2
7	+	M2	M6	A	1,M6
				ST	1,M7
				L	1,M7
8	=	COST	M7	ST	1,COST

図 8.10 コード生成の例

解決する．その理由は，第1の最適化で無駄をなくしておくと，それらについては第2の最適化で時間を費さずにすむからである．

8.1.5.1 機械に独立な最適化

第1の問題は機械に独立な最適化の1つである．1つの文の中で2回以上同じ部分式(共通項)が出てきたら，行列の欄を省略し，それらに対する参照はすべて，保存されている部分式を参照するように変更する(図 8.11)．その結果生成されるコードは図 8.12 の中央の列に示されたようになり，図 8.10 より改良されたものとなっている．

その他の機械に独立な最適化には次のようなものがある：

1. 演算数が両方共定数であるときはその演算をコンパイル時に行なうこと．

共通項のある行列				共通項を削除した後の行列			
1	−	START	FINISH	1	−	START	FINISH
2	*	RATE	M1	2	*	RATE	M1
3	*	2	RATE	3	*	2	RATE
4	−	START	FINISH	4			
5	−	M4	100	5	−	M1	100
6	*	M3	M5	6	*	M3	M5
7	+	M2	M6	7	*	M2	M6
8	=	COST	M7	8	=	COST	M7

図 8.11 共通項の削除

最適化された行列				第1回		改良されたコード		
				L	1,START	L	1,START	
1	−	START	FINISH	S	1,FINISH	S	1,FINISH	M1 → R1
				ST	1,M1			
				L	1,RATE	L	3,RATE	
2	*	RATE	M1	M	0,M1	MR	2,1	M2 → R3
				ST	1,M2			
				L	1,=F'2'	L	5,=F'2'	
3	*	2	RATE	M	0,RATE	M	4,RATE	M3 → R5
				ST	1,M3			
4								
				L	1,M1			
5	−	M1	100	S	1,=F'100'	S	1,=F'100'	M5 → R1
				ST	1,M5			
				L	1,M3	LR	7,5	
6	*	M3	M5	M	0,M5	MR	6,1	M6 → R7
				ST	1,M6			
				L	1,M2			
7	+	M2	M6	A	1,M6	AR	3,7	M7 → R3
				ST	1,M7			
				L	1,M7			
8	=	M7	COST	ST	1,COST	ST	3,COST	
					80バイト		36バイト	

図 8.12 機械に関係する最適化を行なったコード

2. ループの中で変化しない変数の条件をループの外に出すこと.
3. ブール演算式計算の手順を最小とするためにブール代数の性質を利用すること.

これらは第2部でより詳細に論じられる. ここでは行列は**機械に独立な**形で最適化することが可能であり, 行列を用いてコードを生成する前にその最適化が

なされなければならないことを強調しておく．この処理は最適化段階と呼ばれ，コード生成とは論理的に区別される．

8.1.5.2 機械に関係する最適化

第2の問題は，最適化した行列から生成され得る2種のコードを比較するとよく理解できる．

図 8.12 は共通の部分式（M4）を前述の方法で省いて最適化した行列を示している．行列の各欄の右には**図 8.9** に定義された演算子を用いて生成したコードが示されている．3番目の列は使用する記憶域が小さく，適切な命令を使用しているので高速で，前よりはるかに優れたコードを示している．どのようにしてこのように**より効率の良い**コードを得たのであろうか．それは；

1. IBM 360 の 16 個のレジスタを可能な限り有効に使用して一時的な記憶域とし，必要なとき以外中間結果を主記憶に格納しない．このことによりロードと格納が 14 回から 5 回に減った．
2. 可能な限り IBM 360 の**より短かいより高速**の命令を使用した（Mの代りに MR を使用した）．

この例の機械に関係する最適化では，プログラムの所要の記憶容量と目的プログラムの実行時間が共に 1/2 に減少している．機械に関係する最適化は通常コード生成の間に行なわれる．前節で述べたコード生成ルーチンは，条件マクロ擬似命令を持つ複雑なマクロ・プロセッサとみなすことができる．このようにすると機械の構成や一時記憶の内容に応じて生成する命令を変更することができる．この方式によって，コード生成段階で機械に関係する最適化を行なうことができる．

8.1.5.3 アセンブル段階

本節の始めで‘コンパイラの役割りは原始プログラムに等価な**機械語**を生成することである’と述べた．これまでの例ではアセンブラ語を生成している．リテラルや記号番地は人間にとっては理解しやすいが，コンパイラは実際の機械語を作り出さねばならない．

コンパイラは，リテラルと変数をそのままで参照するのではなく，これらの

ために記憶域割付けルーチンによって割付けられた実際の主記憶内の記憶場所を参照したコードを生成することができる．しかしながら，ラベルは最終的にコードが生成されるまで値を割当てることができない．したがって，コード生成に関しては次の手順が必要である：

1. コードの生成.
2. ラベルを定義し，すべての参照を解決する.

これらは論理的に別個のものであり，しばしば別々に組込まれるので，本書では(1)コード生成段階と，(2)アセンブル段階とに分割する．アセンブル段階はアセンブラのパス２に類似している．

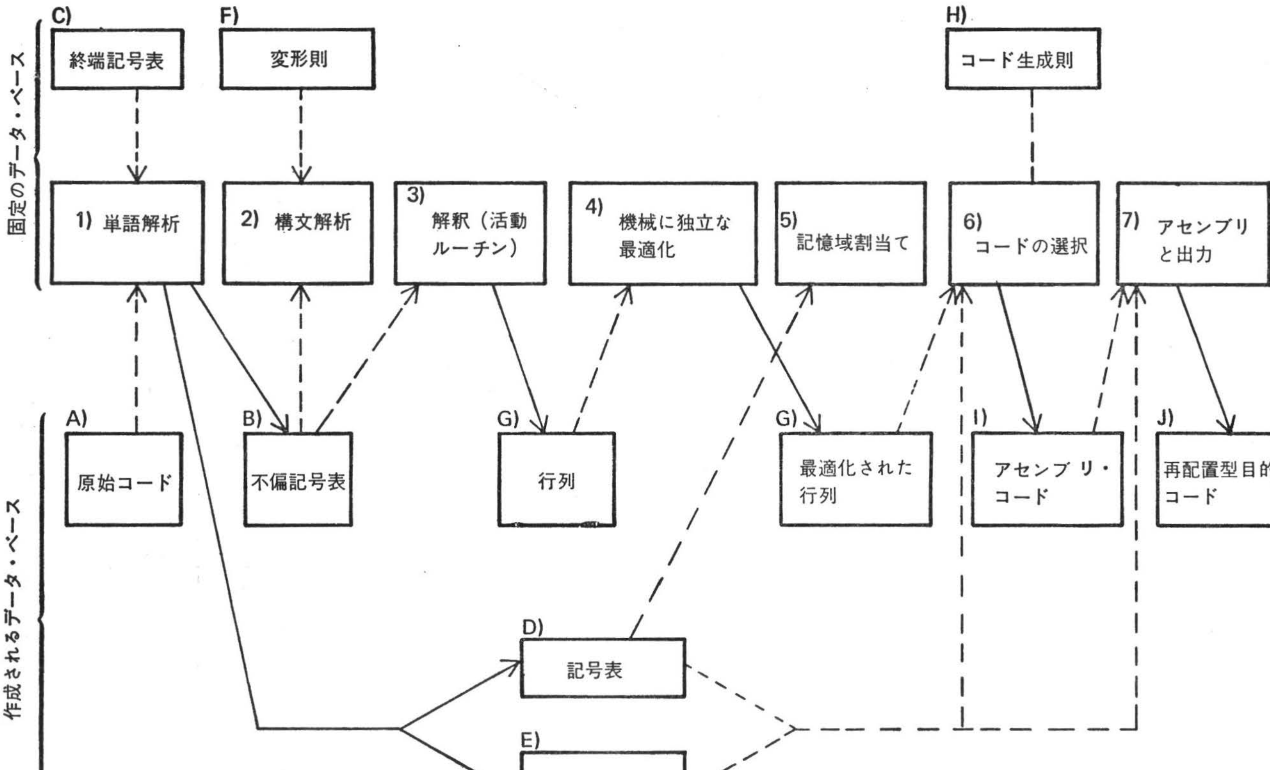
8.1.6 コンパイラの一般的なモデル

簡単な PL/I プログラムのコンパイルを解析した結果，以下のような７つの論理的に分離した問題点を見出した（**図 8.13** 参照）：

1. **単語解析**——基本要素の認識と不偏記号の生成.
2. **構文解析**——変形則による基本的な構文の認識.
3. **解　釈**——正しい意味の定義，活動ルーチンによる行列と表の生成.
4. **機械に独立な最適化**——より最適化された行列の生成.
5. **記憶域の割付け**——記号およびリテラル表の修飾，コード生成段階で動的な記憶域の割付けを行なうコード生成ができるように，また，アセンブル段階で静的記憶域を適当に確保するように行列の欄を構成する.
6. **コード生成**——マクロ・プロセッサを使用してより最適化されたアセンブリ・コードを生成する.
7. **アセンブルとコードの出力**——記号番地（ラベル）を解決し，機械語を生成する.

これらはモデル・コンパイラの７つの段階の名称である．段階１から４は機械に独立で，言語に依存する．段階５から７は機械に依存し，言語には関係しない．効率向上の理由により，実際の組込みではこれらの段階のコードは別個のモジュールとされないこともある．

コンパイラの評価は生成されるコードだけでなく，コンパイラの使用する記憶容量，および実行時間も併せて行なわれる．しかしながら，これらの項目に



実線はデータの作成を示し，点線は参照を示す。コンパイラの段階の間はデータ・ベースを用いて通信する。

図 8.13 コンパイラの構成

関する最適性は互に相反するものである．同様にコードの最適性は，通常，コンパイラ自身の複雑さ，大きさ，および実行時間に反比例する．これらが本章を通じて留意しておかなければならない点である．

また，各段階の間で通信を行なうためにコンパイラが使用するデータ・ベースとして次のものを考えていた：

A．**原始コード**——例では**図8.1**の簡単な PL/I プログラム．

B．**不偏記号表**——プログラムの中に現われたトークンの全部または一部の一覧表．単語解析により作られ，構文解析および解釈に使用される（**図8.3**）．

C．**終端記号表**——言語のすべての見出し語と特殊記号を記号で示した固定の一覧表（**図8.3**の不偏記号によって指される）．

D．**記号表**——プログラム中のすべての変数（例では4個），一時記憶域，およびこれらを参照したり記憶域を割付けたりするのに必要な情報を格納する．単語段階により作られ，解釈段階と記憶域割付け段階で修飾され，コード生成段階とアセンブル段階で参照される（**図8.8**）．表は原始プログラムの実行中に使用するために，コンパイラが作ったすべての一時記憶域に関する情報（たとえば，臨時的な行列の欄）なども含む．

E．**リテラル表**——プログラム中のすべての定数（例では2個）を格納する．Dと同様にして作られ使用される．

F．**変形則**——構文上の構造を見出すために，不偏記号の並びと一致をとることを目的とした書式により決定規則を納めた固定の表．

G．**行　列**——プログラムの中間形式として，活動ルーチンにより作られ，最適化され，その後コード生成のために使用される（**図8.6**）．

H．**コードの生成則**——コード原形の定義を示した固定の表．行列の各演算子に対してコードを定義する欄が1つある（**図8.9**）．

I．**アセンブリ・コード**——コード生成により作られ，アセンブル段階の入力となるプログラムのアセンブラ語版．

J．**再配置型目的コード**——直ちにローダの入力となるアセンブル段階の最終出力．

これらの段階とデータ・ベースおよびこれらの関係は**図8.13**に要約されて

いる．この図は第2部で述べるモデル・コンパイラの構成を表わし，本章の第3部でも使用される．第2部および第3部を読むとき，読者は復習のためにこの図を参照してほしい．

第 2 部

8.2 コンパイラの段階

本章の第2部ではコンパイラ・モデルの7つの段階を詳細に調べる．各段階（**図 8.13**）は出てきた順番に述べられ，データ・ベースについては，コンパイラがそれらを作ったり最初に参照するときに論ぜられる．効率を向上させるため，または言語の特徴を実現するために，実際の組込みにおいてはここで示すより複雑な，あるいは，場合によってはより簡単なデータ・ベースやアルゴリズムを使用することがある．たとえば，FORTRAN や COBOL 型の言語では，PL/I や ALGOL 型の言語のように記号表の中に多くの属性を必要としない．この場合には，ここで示すアルゴリズムは最適なものとはいえない．しかしながら，これらはコンパイラ設計の基本的な道しるべである．読者が独自のコンパイラを組込もうとするときにはデータ・ベースを拡張して例外事項や特例を容易に含めることができるであろう．

8.2.1 単語段階

8.2.1.1 役割り

単語（解析）段階の3つの役割りは次のとおりである：

1. 原始プログラムを言語の基本要素またはトークンに分解する．
2. リテラル表と記号表を作る．
3. 不偏記号表を作る．

8.2.1.2 データ・ベース

上の仕事は5つのデータ・ベースの処理を含む．これらは，

1. **原始プログラム**——プログラムの原形．文字列としてコンパイラに与え

られる.

2. **終端記号表**――各終端記号（たとえば，算術演算子，見出し語，非数値記号）に対して１つの欄を持つ固定の表．各欄は終端記号，分類の表示（演算子，区切り文字），および優先順位（後の段階で使用される）から成っている（第７章問１を参照）.

記号	表示子	順位 †

終端記号表の欄

3. **リテラル表**――原始プログラムの中で使用されたすべてのリテラルを示すために単語段階により作られる．各リテラルに１つの欄があり，値，属性，実行時にリテラルを格納する場所を示す番地（後の段階で記入される），およびその他の情報（たとえば，組込みによっては，プログラムの中で使用されるリテラルと，BINARY FIXED (31) の表現中のリテラル 31 のようにコンパイラによって使用されるものとの区別をしたいことがある）が記入される．データの型や精度などの属性はリテラル自身から求めることができ，単語段階で表に記入される.

リテラル	基数	尺度	精度	他の情報	番地

リテラル表の欄

4. **記号表**――原始プログラムの中で使用されるすべての記号を記述するために単語段階によって作られる．各記号に対して１つの欄が作られる．単語段階は欄を作り，記号の名前をその欄に記入する．多くの言語において記号は 1〜31 文字の長さであるので，単語段階は記憶域の効率を上げるために記号表にはポインタのみを記入することがある．このポインタは名前表の名前を指す．後の段階で各記号のデータ属性や番地が記入される.

名前	データ属性	番地

記号表の欄

† 実線で囲んだ部分は現在論じている段階または前に論じた段階で記入される部分を示す．点線は後の段階で記入される部分を示す.

5. **不偏記号表**——プログラムを個々の文字で表わさず一連のトークンで表現するために単語段階で作られる．（原文中の空白やコンマは不偏記号の表現の中に表われず以後の段階でも使用されない．プログラム中のすべてのトークンに対しては対応する不偏記号が１つずつ存在する.）各不偏記号はトークンが記入されている表の識別符号（たとえば，表またはコードへのポインタ）およびその表の中の欄への指標を持っている．

表	指標

不偏記号表の欄

8.2.1.3 アルゴリズム

単語解析アルゴリズムの第１の役割りは入力文字列を調べてトークンに分解することである．第２は表の中に適当な欄を作ることである．**トークン**とは言語の基本要素を表わす入力記号列の部分である．トークンは単純な文字のみを含むことができ，他のトークンを含んではならない．コンパイラの後の段階ではトークンは処理の最小単位である．単語段階とアセンブル段階の出力ルーチンのみがこの要素を文字として処理する．不偏記号は構文解析における終端記号である．

この段階を組込む方法は多数ある．その１つを次に述べる．**図 8.15** は PL/I プログラム例の単語段階の結果を示す．

入力記号列は区切り文字でトークンに分割される．区切り文字は終端記号表の特別な欄の内容で示される．原文の文字が読込まれ，正当性が検査され，それらが区切り文字かどうか判定される．相続く区切り文字でない文字は１つのトークンとされる．区切り文字にはさまれた区切り文字でない文字の並び，および空白以外の区切り文字がトークンである．空白は区切り文字となるがトークンとはみなされず無視される．

単語段階は３種のトークン，終端記号，記号となる可能性のあるもの，およびリテラルを認識する．すべてのトークンは，まず，終端記号表の欄と比較される．一致するものが見つかるとトークンは終端記号であるとされ，単語段階は 'TRM' 型の不偏記号を作り，これを不偏記号表に挿入する．トークンが終端

記号でなければ，単語段階はさらに記号となり得るものかまたはリテラルかの区別を行なう．記号を形成する単語の規則を満たすトークンは“記号となり得るもの”として区別される．PL/I ではこれらは英文字で始まり，それに最大30文字までの英数字か下線かが続く．トークンがこれらいずれかに適合しないときは誤りであり，その旨表示される．

トークンが“記号になり得るもの”とされた後に記号表が調べられる．このトークンが表になければ新しい欄が作られる．記号についてこの段階でわかっている属性は名前だけであり，表には名前だけが記入される．残りの情報は後の段階で見出され記入される．記号表の欄が作られても作られなくても，単語段階は‘IDN’型の不偏記号を作り，不偏記号表に挿入する(図8.15の例を参照).

数，“′”でくくられた文字列，および自己定義のデータは“リテラル”として取り扱われる．トークンがリテラルとみなされると，リテラル表が調べられる．リテラルが表の中になければ，新しい欄が作られる．リテラルの場合には，記号の場合とは対照的に，単語段階はそれを表わしている文字列を調べて属性および内部表現を決定することができる．したがって，リテラル表の中に作られる新しい欄はリテラルとそのすべての属性から構成される．新しい欄が作られるか否かに拘らず，‘LIT’型の不偏記号が作られ，不偏記号表に記入される．

本節で述べた単語段階では原始コードを最初から最後まで調べ，完全な不偏記号表を生成する．このほかに，構文解析で次のトークンが必要になったときのみ単語解析が呼出され，原始コードを調べるという方法もある．

〈記号〉 ::= 〈文字〉 | 〈文字〉 | 〈文字数字〉$_{1}^{30}$
〈文字数字〉 ::= 〈文字〉 | 〈数字〉
〈文字〉 ::= A | B |…| Z
〈数字〉 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
〈リテラル〉 ::= 〈数字〉$_{1}^{15}$
〈項〉 ::= 〈演算子〉 | DECLARE | END | PROCEDURE | RETURN | (|) | ; | : | , |
　　STAIC | FIXED | BINARY | b̸ |
〈演算子〉 ::= + | − | * | ÷ | ** | =

図 8.14 単語規則のBNF表現

8.2.1.4 例

図 8.14 は記号の3つのクラス（記号，リテラル，終端記号）の形式を定義している．各々は不偏記号（**IDN, LIT, TRM**）となる．各トークンに対して不偏記号が作られ，まだ表に登録されていない記号またはリテラルに対しては

終端記号表

	記号	区切りか	その他
1	:	Yes	
2	;	Yes	
3	(	Yes	
4	)	Yes	
5	,	Yes	
6	b̸	Yes	
7	PROCEDURE	No	
8	DECLARE	No	
9	RETURN	No	
10	END	No	
	+		
	−		
	⋮		

不偏記号表

種類	指標	トークン
IDN	1	WCM
TRM	1	:
TRM	7	PROCEDURE
TRM	3	(
IDN	2	RATE
TRM	5	,
IDN	3	START
TRM	5	,
IDN	4	FINISH
TRM	4	)
TRM	2	;
TRM	8	DECLARE
TRM	3	(
IDN	5	COST
TRM	5	,
IDN	2	RATE
⋮		⋮

記号表

	名前	属性
1	WCM	後の段階で記入
2	RATE	
3	START	
4	FINISH	
5	COST	

リテラル表

リテラル	基数	尺度	精度	その他	番地
31	DECIMAL	FIXED	2		
2	DECIMAL	FIXED	1		
100	DECIMAL	FIXED	3		

図 8.15 単語段階――例の終端記号，記号，不偏記号表

記号表やリテラル表の中に1つの欄が作られる（例として**図8.15**参照）.

8.2.2 構文段階

構文（解析）段階の機能は，言語の主要構造を認識し，これらの構造に対して中間形式または行列を生成する適当な活動ルーチンを呼出すことである．コンパイラによっては，この段階は各構造を認識する1つの大きなプログラムによって組込まれることがある．本書ではより一般的な組込みを示そう．構文段階は，言語の主要構造を決定し，各構造に対してとるべき適当な活動を決定する法則または変形則に対する解釈ルーチンである．

変形則は原始言語の構文に依存し，したがって，原始言語が変更されれば規則も変更される必要がある．変形則はその内容が問題であり，その形式は全く任意でよい．構文段階でどれだけのことを行ない，解釈段階の活動ルーチンでどれだけのことを行なうかの問題については後で述べることにする．効率を上げるために多くのコンパイラは構文段階では解釈ルーチンを持たず，固定コードで処理を行なっている．コンパイラをもっと効率良くしようとすれば変形則を“コンパイル”することもできる．組識的に構文段階を組込む方法は，変形則を用いてその処理を記述することである．次に設計者は，必要ならば，変形則を1つのプログラムへ手で翻訳すればよい．

8.2.2.1 データ・ベース

不偏記号表（UST）—— 単語段階で作られ不偏記号形で表現した原始プログラムを格納している．この表は構文段階および解釈段階でスタックへの入力源として使用される． UST の各記号は1回だけスタックに入る．

表	指標

不偏記号表の欄

スタック——スタックは現在構文段階および解釈段階で処理中の不偏記号の集合である．スタックへの追加や削除はこれを使用する段階においてなされる．スタックは**最後に入ったものが最初に出る**（Last In First Out: LIFO）という記憶域である．本書ではスタックはその先**後入－先出**，頭がページの下の方

に向かっているとして表わす．“スタックの先頭”とは最も新しく作られた欄をいい，“スタックの底”とは最も古い欄をいう．スタックは不偏記号表と結合して構成することもでき，またその逆の場合もあり得る．

変形則——原始言語の構文規則は変形則の表に格納されている．構文段階は変形則によって動作するインタプリタである．変形則の一般形は；

ラベル：古いスタックの先頭／活動ルーチン／新しいスタックの先頭／次の変形則

ただし，次の規約が使用される：

1. ラベル——任意．
2. 古いスタックの先頭——スタックの先頭と比較される．
    a. 空白または null——スタックに何が入っていてもつねに一致するとみなされる．
    b. 空白以外——次のいずれかに属す．
        1) 記号やリテラルなどの〈構文形〉——この型のすべての不偏記号と一致するとみなされる．
        2) 〈任意〉——任意の不偏記号と一致するとみなされる．
        3) 見出し語の記号表現，“PROCEDURE”，“IF”，“：”など——この見出し語に対する不偏記号とだけ一致するとみなされる．
3. 活動ルーチン——古いスタックの先頭がスタックの先頭に一致したとき呼出される．
    a. 空白または null——活動ルーチンは呼出されない．
    b. 活動ルーチンの名前——そのルーチンを呼出す．
4. 新しいスタックの先頭——活動ルーチンが実行された後にスタックの先頭が変更される．
    d. 空白または null——変化なし．
    c. ″——スタックの先頭を抹消する（すなわち，一致したパターン）[2]．
    b. 〈構文形〉，見出し語またはスタックの項目（Sm）——古いスタックの先頭を抹消し（2で一致したパターン），この項で置換える．
    d. *——不偏記号表から次の不偏記号を求めてスタックの先頭に置く．

2　これは必ずしもスタックの要素をすべて抹消するのではない．変形則の古いスタックの先頭にある一致した部分のみが抹消される．

5. 次の変形則
   a. 空白または null——次に続く変形則を解釈する.
   b. n——変形則 n を解釈する.

変形則の中でスタックを示す2つの部分を表わすのに最も古い項目を左端に，最も新しい項目を右端に置いている．$S_m\,S_{m-1}\,\cdots\,S_2\,S_1$ は長さ m の部分を示す．S_1 はスタックの先頭を，S_2 はスタックの2番目の項目…を示す．S_n は新しいスタックの先頭の部分から n 番目の項目を示す．変形則を効率良く組込むために，実際のスタックの要素としては，不偏記号が用いられるが，本節の説明においては，古いスタックの先頭の部分には終端記号そのものを用いて表わすことにする．

8.2.2.2 アルゴリズム

構文解析のアルゴリズムは次のとおりである：

1. 古いスタックの先頭と現在のスタックの先頭の間で，変形則の一致するものが見出されるまで順番に調べる．
2. 一致するものが見出されると，活動部で指定された活動ルーチンを左から右への順に実行する．
3. 制御が構文解析ルーチンに戻されると，スタックの先頭を修飾して新しいスタックの先頭部とする．
4. 次の変形則部で指定されている変形則から開始してステップ1を繰り返す．

8.2.2.3 例

第1部の例題を詳しく解析して解釈段階を説明する．ここでは次の3つの変形則のみを説明する：

```
    | | *** |
2:  〈記号〉: PROCEDURE / bgn_proc / S1 **** / 4
    〈任意〉〈任意〉〈任意〉/ error / S2 S1 * / 2
```

これら3つの変則形は次節の例において定義される最初の3つである．構文段階を実行してこれらを解釈すると；

1. UST からスタックへ最初の3つの不偏記号を入れる．

2. 先頭の3要素が"〈記号〉：PROCEDURE"であれば，活動ルーチン bgn_proc を呼出し，ラベルと"："を抹消し，USTからスタックへ次の4つの不偏記号を移し，変形則4へ行く．先頭の3要素が"〈記号〉:PROCEDURE"でないときは，活動ルーチン error を呼出し，スタックから3番目の不偏記号を取り除き，UST から1つの記号を求めてスタックに入れ，変形則2へ行く．

この変形則は，プログラムは'〈ラベル〉：PROCEDURE' で始まらなければならないということを述べている．PROCEDURE 文を認識すると，活動ルーチン bgn_proc は〈ラベル〉をラベルとして分類する．次に構文段階でラベルと"："を取り除き，次の4つのトークンを取り出す．そして変形則4を解釈すると手続き本体の解析が始まる．最初の文が'〈ラベル〉:PROCEDURE'でなければ，error を呼出して誤りを通知し，診断メッセージを印刷する．次に S_3 が消去され，スタックに新しいトークンが置かれて変形則2が再び試みられる．これは，'〈ラベル〉：PROCEDURE' と一致するまで，または，UST のすべての記号が調べつくされるまで行なわれる．

8.2.3 解釈段階

解釈段階は，構造が構文段階で認識されたとき呼出されるルーチンの集合である．活動ルーチンと呼ばれるこれらルーチンの目的は，原始プログラムの中間形式を作り，記号表に情報を追加することである．

構文段階と解釈段階を分割するのは論理的な割り方である．前者は構文（たとえば文や句などの意味を持つ節）を認識し，後者は正確な意味を解釈して行列や記号表を作る．実際にはどこが構文認識の終りで，どこからが意味解釈の始まりかは定かではない．言葉を換えていうと，意味づけを行なう節の大きさは任意に決定してよいということである．このことは算術式を処理する問題を例として説明する．1つの算術式の全体は構文の1つの構造であると考えることができる．したがって，これの認識は構文段階の問題であり，解析は活動ルーチンの問題である．しかし，各部分表現（2つの演算数および演算子）は同様に構造であると考えることもでき，構文段階で認識する必要がある．最初の場合の活動ルーチンは式全体のために行列の欄を作る．第2の場合の活動ルーチ

ンは部分式に１つの欄を作るだけでよい．どれだけの処理を構文段階で行ない，どれだけの処理を活動ルーチンを行なうかの分割が問題である．

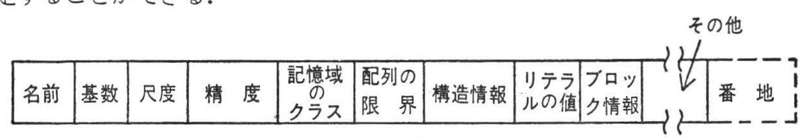
本節以降の例では変形則の機能と活動ルーチンの機能との分割を，解析する文の種類によって変化させている．このようにしてこの問題の様々な解決法を理解してもらおうと考えている．一般にコンパイラ作成者は仕事が効率良く行なえるよう，すなわち，２つの段階の複雑さを最小とするように分割を行なうだろう．構文段階では構造の認識を，解釈段階では行列の欄を直接作り出したり記号表を修飾する特定の役割りを与えることが一般的である．

8.2.3.1　データ・ベース

不偏記号表

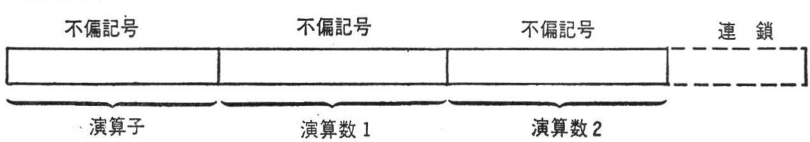
スタック——構文段階および解釈段階で解析されているトークンを格納する．

記号表——単語段階に入るとき，初期化される．単語段階では記号（または文字列へのポインタ）が名前の部分に記入される．解釈段階ではすべての属性が記入される．記憶域割付け段階では番地が記入される．この情報は後の段階で記憶域の割付けを行なうこと，および，この欄に関連する記憶域を参照する適当なコードを生成するために使用される．大半の属性は宣言文の解析により決定することができる．



リテラル表の欄

行列——プログラムの第１次の中間形式を使用する理由は２つある：(1)コンパイラの最初の４つの段階を機械に独立とする．(2)コード生成段階の前にプログラムの最適化を可能にする．行列の欄の簡単な形は次の３項子である．第１要素は終端記号または演算子を示す不偏記号であり，他の２つの要素は引数を示す不偏記号である．コード生成段階は行列を用いて実行され，適当な目的コードを生成する．



行列の欄

行列の欄には欄を連結するために使用される部分が作られることもある．この欄は最適化段階で欄を追加したり削除したりするのに使用される．

行列の中に記入されるものは不偏記号であるが，例ではわかりやすくするために記号自身を使用している．行列の演算子には，原始言語や機械語に直接相当するもののないことがある．たとえば，複合文 IF THEN.........ELSE などは行列の複数の欄に翻訳され，それらは各々幾つかの機械命令に翻訳される．

行列の演算数は IDN, LIN, TRM, および MTX 型の不偏記号である．不偏記号 MTX n は行列の第 n 欄を示し，記号表の一時記憶域欄を指している．

一時記憶域表（記号表の一部として組込まれることもある．）――解釈段階は一時記憶域表に MTX 記号に対応した値に関するすべての情報，すなわち，行列の欄を計算した結果のデータの型や精度，それが使用される原始文などの属性を記入する．この情報は記憶域割付け，およびそれを参照するコードの生成において使用される．

MDXN	基数	尺度	精度	記憶域の クラス	その他	番地

一時記憶域表

8.2.3.2　アルゴリズム

この段階に対しては１つにまとまったアルゴリズムはない．その理由はこの段階は構文解析によって呼出されたとき特定の役割りを果たす個々の活動ルーチンが集ってできているからである．特に，これらのルーチンは；

1. 必要とされる付加的な解析を行なう――このため活動ルーチンは必要と考えられるとき，スタックに記号を加えたり削除したりすることができる．
2. 行列に新しい欄を作ったり，記号表にデータ属性を追加したりする――前の場合には，ルーチンは適切な演算子と演算数を決定し，行列の中に記入できなければならない．後の場合には，どの属性が宣言されたかを正確に決定し，記号表に記入しなければならない．いずれにせよ活動ルーチンの複雑さはどこまで変形則によって処理されているかに依存し，また，その逆のこともいえる．

次の例は幾つかの複雑な活動ルーチンと幾つかの単純な活動ルーチンを示し，何がどの段階に含まれなければならないかという問題にははっきりした答えがないことを説明している．

8.2.3.3 例

ここでは第1部の例題をコンパイルするときの構文段階と解釈段階の働きを例として示す．例題をより理解しやすくするために PL/I のごく小さなサブセット (MINI—PL/I) だけを処理するコンパイラを考えている．

MINI—PL/I は次のような言葉で定義できる．

文の種類は5種類である．その書式は次のとおりである：

(1) ラベル：PROCEDURE（引数のリスト）

(2) DECLARE AB の並び

A：〈記号〉または（〈記号〉のリスト）

B：FIXED BINARY（精度）STATIC

AB の節は“,”で区切られ，文は“；”で終了すること．

(3) 〈記号〉＝正しい算術式；

(4) RETURN（引数）；

(5) END;

前の例は MINI—PL/I の正しいプログラムである．上の規則に適合しないものは正しくないプログラムである．正しい MINI—PL/I プログラムは書式 (1) の文で始まり，文 (5) で終らなければならず，その間に文 (2)，(3) および (4) のみを持つ．

このような言葉による定義は不正確であるから，この定義によってコンパイラを作るのは困難である．たとえば引数のない PROCEDURE 文は正しいであろうか．そこで，**図8.16**で BNF を用いて形式的に MINI—PL/I を定義する．この定義が理解できないときは第7.5節と第7章の問1とを復習してほしい．BNF 定義は単語段階において定義された3つのクラスの要素を用いている．またこれらの規則には演算子の優先順位が存在しないことも注意してほしい．これらのことは第7章の問1および問7に含まれている．

原始言語を形式的に定義すれば，認識したい構文上の構造（節）を決定する

```
<手続き>::=<手続><本体><終>                                手続き
<手続>::=<記号>：PROCEDURE (<変数>);                       手続き文
<終>::=END；                                               終了文
<変数>::=<記号>|<記号>，<変数>
<本体>::=<文>|<文><本体>                                   手続きの本体
<文>::=<代入>|<復帰>|<宣言>                                文
<代入>::=<記号>=<式>;                                    ⎫
<式>::=<リテラル>|<記号>|<式><演算子><式>|               ⎬ 代入文
          (<式>)                                         ⎭
<復帰>::=RETURN (<記号>);                                  復帰文
<宣言>::=DECLARE <宣言要素>;                              ⎫
<宣言要素>::=<変数リスト><属性リスト>|                    ⎪
               <変数リスト><属性リスト>，<宣言要素>       ⎬ 宣言文
<属性リスト>::=FIXED BINARY (<リテラル>) STATIC           ⎪
<変数リスト>::=<記号>|(<変数>)                            ⎭
終端記号の種類：(構文段階のための記号――これらは単語段階で決定される.)
<記号>     ⎫
<リテラル> ⎬ 図 8.41 で定義されている.
<演算子>   ⎭
```

図 8.16　MINI—PL/I の BNF 定義

```
1:   //***/
2:   〈記号〉: PROCEDURE / bgn_proc/ S1 **** / 4              ⎫ スタックの初期化
     〈任意〉〈任意〉〈任意〉 / error / S2 S1 * / 2           ⎭ 手続きの判定
4:   (〈記号〉,〈任意〉 / arg / S4 S1 ** / 4                  ⎫
5:   (〈記号〉);/ arg / " ** / 7                             ⎭ 引数表
     〈任意〉〈任意〉〈任意〉〈任意〉 / error / S2 S1 / 7
7:   DECLARE 〈任意〉 / / S1 ** / 14                         ⎫
     RETURN 〈任意〉 / return / S1 *** / 5                   ⎪ 宣言か
     〈記号〉= /op_prec/ / 12                                 ⎪ 復帰か
     PROCEDURE END; /end_proc/ " *** / 2                     ⎪ 代入か
     〈任意〉〈任意〉 / error / S1 * / 7                      ⎭ 終了か
12:  〈記号〉=〈任意〉 ; / assign / " ** / 7                   ⎫
     〈任意〉〈任意〉〈任意〉〈任意〉 / error / S2 S1 / 7      ⎭ 代　入
14:  〈記号〉FIXED BINARY / var_name base_scale / " ***** / 18 ⎫
15:  (〈記号〉, /var_name / S3 ** / 15                        ⎬ 名前，基数，尺度の宣言
     (〈記号〉) / / S2 ** / 14                                ⎪
     〈任意〉〈任意〉〈任意〉| / error / S2 S1 / 7              ⎭
18:  (〈リテラル〉) STATIC , / precision_class / " *** / 14    ⎫
     (〈リテラル〉) STATIC ; / precision_class / " ** / 7      ⎬ 精度，クラスの宣言
     〈任意〉〈任意〉〈任意〉〈任意〉〈任意〉 / error / S2 S1 / 7 ⎭
```

図 8.17　構文段階の変形則――MINI—PL/I

ことができる．次に各節に対応した活動ルーチンを定め，構文構造を認識し解

析して適当な活動ルーチンを呼出す変形則を書かねばならない．実際には，節と活動ルーチンを定義し，変形則を記述する仕事は互に関連し合うものである．すなわち，最初1組の活動ルーチンを仮定し，変形則を記述して行くにつれて活動ルーチンを変更したり追加したりする．**図 8.17** は MINI—PL/I に対して選んだ変形則の一覧表である．第 8.2.3.2 節ではこれら変形則の最初の3つを説明した（これらの説明が理解できなかった読者はもう一度説明を読み直してほしい）．

図 8.18 は選ばれた活動ルーチンの一覧表である．これらの活動ルーチンの複雑さは様々である．たとえば，ルーチン var_name はきわめて単純である．このルーチンは 1つの記号を 表に追加することだけを 行なう．活動ルーチン basescale および precision_class はこの表に各記号に対する属性を 記入する．ルーチン prcision_class は記号の群に対する属性を挿入する最後のルーチンであるので使用後にこの表を消去する．

1 arg—'arg<記号>' を行列に入れる．
2 assign—'=<記号><任意>' を行列に入れる．
3 base_scale—'基数=BINARY' を適当な記号表の欄に入れる．
4 bgn_proc—'procbgn<記号>' を行列に入れる（記号は記号表の中のラベルである）．
5 end_proc—'procend' を行列に入れる．
6 error—誤りメッセージとスタックの先頭とを印刷する．
7 op_prec—算術式を解析し，演算子の優先順位の規則を用いて行列の欄を生成する．<結果>をスタックに残す．
8 precision_class—精度と 'STATIC' を適当な記号表に記入する．
9 return—'rtn' を行列に入れる．
10 var_name——時記憶域表に名前を入れる．

図 8.18 活動ルーチン

ルーチン op_prec は，1つの算術式全体を解析するので，もっと複雑である．このルーチンは，1つの演算子に適当な演算数を対応させ，次に実行されるべき順序で行列に演算を記入するために演算子の優先順位（第7章問1および7で説明されている）の規則を使用する．このルーチンは“；”を見つけると解析を停止し，行列の最後の行と“；”をスタックに残す．ルーチン var_name とルーチン op_prec は構文解析と解釈の段階の間における仕事の分割の例である．

ここで第1部の例題を翻訳する変形則が，単語段階で作られたデータ・ベースから中間形式（**図 8.19** 参照）を作り出す過程を追ってみよう．読者は**図 8.19** をぜひ自分で検証してみてほしい．

図 8.19 では読者の便宜を計って，不偏記号表の中のすべての不偏記号，変形則，スタックおよび行列は記号で示されている．実際のコンパイラにおいては，これらはすべて | 表 | 指標 | の形で示される．また，簡単のためにPROCEDURE を PROC と省略する．

図 8.19 は，これら2つの段階で例題のコンパイルを行なうときのステップを寸分違わず示している．**図 8.20** は，後の4つの段階のために行列，記号表，およびリテラル表を作るときの最初の3つの段階の間の相互関係を要約している．

8.2.4 最適化段階

第1部では，コンパイラによって行なわれる2種の最適化，機械に関係するものと機械に独立なものとの区別を行なった．

機械に関係する最適化は生成される命令に密接に関係しているので，コード生成段階に組み入れられる．一方機械に無関係な最適化は独立の最適化段階で行なわれる．本節では最適化段階と，第 8.1.5.1 節(第1部)にて述べた通常よく行なわれる4つの型の機械に独立な最適化の技法を論じる．

コンパイラにこれらの最適化法を組入れるか否かを決定するに当っては，これを行なうことによる目的プログラムの効率向上とコンパイル時間や複雑さの増加とをよく勘案しなければならない．この決定はコンパイラが作成されたり使用されたりする環境によって左右される．

8.2.4.1 データ・ベース

行列——これは最適化段階で使用される主要なデータ・ベースである．行列の欄を削除したり挿入したりできるように，各欄に連結情報——前向きおよび後向きのポインタ——を追加する．このようにすることによって，1つの欄が追加されたり消去されたりするとき，行列の欄を並べ直したり場所を変更したりせずにすむ．前向きのポインタを用いることによって，コード生成段階は適切な順序で行列を読出すことができる．後向けポインタを用いることによって，

図8.17の変形則番号	一致	活動ルーチンと操作	行列の欄番号	行列			スタック
1	Yes	何もしない					WCM: PROCEDURE
2	Yes	bgn proc-(WCMは記号表のラベルである) 手続き名を行列に入れる	M1	.procbgn	WCM		PROC (RATE,START
4	Yes	arg-第1引数を行列に入れる	M2	.arg	RATE		PROC (START,FINISH
4	Yes	arg-第2引数を行列に入れる	M3	.arg	START		PROC (FINISH);
4	No						
5	Yes	arg-第3引数を行列に入れる	M4	.arg	FINISH		PROC DECLARE (
7	Yes	何もしない					PROC (COST,
14	No						
15	Yes	var_name －COSTを表に入れる					PROC (RATE,
15	Yes	var_name －RATEを表に入れる					PROC (START,
15	Yes	var_name －STARTを表に入れる					PROC (FINISH)
15	No						

図 8.19 構文段階および解釈段階のステップ

構文／解釈段階：(続き)

図8.17の変形則番号	一致	活動ルーチンと操作	行列の欄番号	行列			スタック
16	Yes	何もしない					PROC FINISH FIXED BINARY
14	Yes	var_name — FINISHを表に入れる					
(14続き)		base_scale-フラグがオンとなっている記号表のすべての欄に対して，尺度＝FIXED，基数＝BINARYと設定する					PROC (31) STATIC ;
18	No						
19	Yes	precision class-フラグがオンとなっている記号表のすべての欄に対して，精度＝31，クラス＝STATICと設定する					PROC COST =
7,8	No						
9	Yes	op_prec -がくるまで算術式を解析して，演算子の優先順位の規則を用いて適当な行列の欄を作る．このためには，終端記号表にある演算子に関して，右および左の優先順位を参照しなければならない	M5	—	START	FINISH	
			M6	*	RATE	MTX5	
			M7	*	2	RATE	
			M8	—	START	FINISH	
			M9	—	MTX8	100	
			M10	*	MTX7	MTX9	
			M11	+	MTX6	MTX10	PROC COST = MTX11;

図 8.19 構文段階および解釈段階のステップ（続き）

構文／解釈段階：(続き)

図8.17の変形則番号	一致	活動ルーチンと操作	行列の欄番号	行列			スタック
12	Yes	assgn-＝, COST, および式の値を行列に入れる	M12	=	COST	MTX11	PROC RETURN (
7	No						
8	Yes	return − rtnを行列に入れる	M13	.rtn			PROC (COST);
5	Yes		M14	.arg	COST		PROC END;
7,8,9	No						
10	Yes	end_proc − .procendを行列に入れる	M15	.procend			不偏記号表終了

図 8.19 構文段階および解釈段階のステップ（続き）

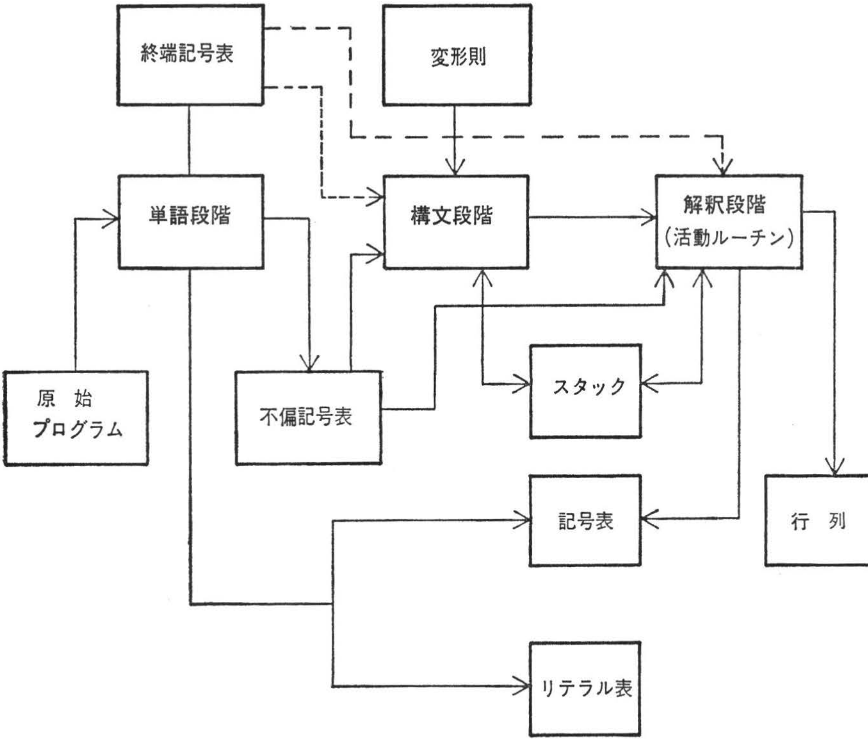


図 8.20 単語段階，構文段階および解釈段階の関係

たとえばDOループの外に計算を移すような最適化が必要なとき，行列の中を逆にたどることができる.

行列の欄は次のようになる：

演算子	演算数 1	演算数 2	前向けポインタ	後向け ポインタ

ここで前向きポインタは行列の次の欄への指標であり，後向きポインタは前の欄への指標である.

記号表——不必要な一時記憶域を消去し，記号に関する情報を求めるために参照される.

リテラル表——ある種の最適化によっては新しいリテラルが作られることがあ

る.

8.2.4.2 アルゴリズム

4つの最適化技法のアルゴリズムについて論じる.

1. 共通項の削除——第 8.1.5.1 節では行列の中から同じ形の欄を消去すると,簡潔で効率の良いプログラムが生成されることを示した.共通項は同一の表現であり,同じ文の中に入っていなければならない.たとえば次のプログラムでは,変数 START が2つの式の計算の間で変更されるので,2番目の START－FINISH に対応した行列の欄を消去することはできない.

```
COST  = RATE * (START - FINISH);
START = START / 2;
COST2 = RATE * (START - FINISH - 100);
```

PL/I などの言語では任意の時点で割込みが発生して変数の値を変更することがあるので,共通項の削除は,割込みルーチンが変数の値を変更しないことをプログラマが規定しているときのみ実行できる.たとえば PL/I プログラムでは ON OVERFLOW (FINISH＝0) と宣言できる.このようにすると,同じ文の中で割込みが発生したとき2つの共通項は異なった値を持つ場合が生じる.

消去アルゴリズムは;

1. 共通項を識別できるような形式で行列に記入し,

2. 2つの共通項が等価であることを認識し,

3. それら一方を削除し,

4. この欄の削除に伴ない行列の残りの部分を変更しなければならない.

1つの文で欄を修飾したならば,このアルゴリズムはその修飾が別の共通項を生み出していないかどうかを確めることが必要である.

図 8.21 はプログラムの幾つかの文に対して作られた行列を示している.共通項削除のアルゴリズムは次のステップを実行する:

1. ＋および＊演算子の演算数をアルファベットの順に並べる——各欄を調べて順番が逆であるものを入換える(図では M3 と M4 の入換えが必要である).

2. 文の境界を明示する——この場合には単に“＝”演算子を用いて分割する.

3. 共通項を探す——M2 と M3, M4, M5 とを比較する．この場合 M3 とのみ一致するので，M3 を削除して M2 と M4 のポインタを変更する．

4. 行列に残っている欄で削除した欄を参照しているものがあれば，これを変更する．たとえば，M3 への参照をすべて M2 に変更する．

5. 新しく共通項が生じていないか調べ直す——変更がなくなるまで 1 から 5 を繰り返す．

6. 不要になった MTX の欄を一時記憶域表から削除する．

原始コード

```
B = A
A = C * D * (D * C + B)
```

最適化を行なう前の行列

M1	=	B	A	0	2
M2	*	C	D	1	3
M3	*	D	C	2	4
M4	+	M3	B	3	5
M5	*	M2	M4	4	6
M6	=	A	M5	5	?

ステップ 1 と 2 の後の行列

M1	=	B	A	0	2	文
M2	*	C	D	1	3	
M3	*	C	D	2	4	
M4	+	B	M3	3	5	文
M5	*	M2	M4	4	6	
M6	=	A	M5	5	?	

ステップ 3 の後の行列

M1	=	B	A	0	2
M2	*	C	D	1	4
M3	*	C	D	2	4
M4	+	B	M2	2	5
M5	*	M2	M4	4	6
M6	=	A	M5	5	?

図 8.21 共通項の削除の例

2. コンパイル時の計算——コンパイル時に定数に関する計算を行なうと，目的プログラムの記憶容量と実行時間を節約することができる．ループの中にこのような算術演算が存在するときには必ずこれを行なっておかなければならない．たとえば次のような文があったとする：

A＝2＊276 / 92＊B

コンパイラは掛算と割算を指定されたように実行することができ，上の式の代りに 6＊B を代入することができる．これによって行列の 2 つの欄を消去する

ことができ，したがって，生成される目的コードの量を少なくすることができる（図 8.22 参照）.

この最適化法のための簡単なアルゴリズムは，まず，行列を走査して2つの演算数が共にリテラルであるような演算子を見つけることである．これが見つかるとその値を計算し，新しいリテラルを作り，元の行を削除し，その行への参照をすべて新しいリテラルに対する不偏記号で置換える．そして再び走査を続けて計算のできる項を探す．

最適化の前の行列

M1	*	2	276
M2	/	M1	96
M3	*	M2	B
M4	=	A	M3

最適化の後の行列

M1			
M2			
M3	*	6	B
M4	=	A	M3

図 8.22　コンパイル時の計算

3.　ブール代数式の最適化——論理演算の長さを短かくするためにブール代数の性質を利用することができる．たとえば，IF a OR b OR c THEN…という文が与えられ，a, b, c が表現であるとする．この場合各表現 a, b, c をつねに判定するコードを生成せずに，もし a が真ならば b OR c を計算しないようなコードを生成する．b についても同様である．

4.　不変項の計算をループの外側に移す——ループの中の計算がループの中で変化しない変数だけに関係しているならば，この計算はループの外へ移すことができる．

これを行なうには行列の欄の順番を一部入換えることが必要である．この型のアルゴリズムでは，次の3つの一般的な問題を解決しなければならない：

a.　不変項計算の認識

```
          DO    I = 1 TO 10;
OUTLOOP:  ⋮
          DO    J = 1 TO 10;
     LA:  A = 10;
     LB:  B = Y(I + 2);
     LC:  C = C + 10;
          END;
          END;
```

このプログラムの場合にはループの外に何が移せるか.

文 LA はつねに同じ結果を生じるのでループの外に出すことができる. LB では添字 I+2 は同じく不変であり, 内側のループの外に移すことができる. Yの値は内側のループの中で変更されるかどうかを調べることも必要である. この例では変更されないので不変項であり, 外へ移すことができる. LC はCを含む計算であり, Cは内側のループで変更される変数であるので外側に移すことはできない. ループの内側でLAより前にAまたはBを参照するものがあれば, これらの算術文を内側のループから外へ移すことができないことに注意してほしい. この場合外側へ移せるのは添字の計算だけである.

b. 不変項の計算を移すべき場所を見つけること

一般にはこの計算をループの直前の位置に移したい. このためには行列を調べてDO ループの開始点を求めなければならない. 解釈段階ではすべてのDO ループの始めで行列の中に DO ループであることを示す欄を置かねばならない. また, 行列の中で後向けポインタを作り, 最適化段階でループの外側へさかのぼれるようにしておかなければならない.

c. 不変項の計算の移動

共通項の削除で論ぜられたのと同じ連鎖の方法を使用して, 行列の元の場所から不変項の計算を削除して適当な場所に挿入する.

8.2.5 記憶域の割当て段階

この段階の目的は次の事柄を実行することである:

1. 原始プログラムの中で参照されているすべての変数に記憶域を割当てる.
2. 中間結果, たとえば, 行列の行の結果など, に対して必要となるすべての一時記憶域に記憶域を割当てる. これらの記憶域参照は解釈段階で確保されるものであり, 原始コードの中には現われない.
3. リテラルに記憶域を割当てる.
4. 記憶域が割付けられ, 所要の場所が初期化(リテラルおよび初期値属性を持つ変数)されるようにする.

これらを行なうためには, 記号表, リテラル表, および行列の中にコード生

成のための情報を挿入しなければならない.

8.2.5.1　データ・ベース

記号表——単語段階によって作られ，解釈段階によってすべての属性が与えられている．記憶域割当て段階はデータを指すすべての記号のために記憶場所を指定する．次の２つの段階でこの情報を使用して記号の参照を生成する．

							初期値	
名前	基数	尺度	精　度	記憶域のクラス	配列の限界	構造情報		番　地

記号表の欄

一時記憶域表（記号表の一部とされることもある）——行列の中の計算の一時的な結果を記憶するために解釈段階で作られる．この表は，多くの情報の書式が同じであるので記号表の一部として組込まれることがある．さらに，この型の記憶域はすべて自動変数として組込まれる．自動変数は手続きが使用されるまでは物理的に割付けられない．他の手続きではこの変数を参照することがないし，その手続きが使用されるまではその領域は必要でない．

M_i	基数	尺度	精　度	AUTOMATIC	その他	番　地

一時記憶域の欄

リテラル表——記憶域割当て段階は，すべてのリテラルに番地を割当て，コー

記憶域のクラス	容　量	
演算子	演算数	演算数

初期化	記号またはリテラル	
演算子	演算数	演算数

記憶域割付けによる行列の欄

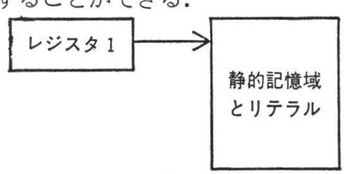
ド生成段階でこの記憶域を割付けるように行列の欄を作る.

行 列――記憶域割当て段階は，コード生成段階で記号，一時記憶域，および リテラルに十分な記憶域を作り出すようにするため行列の中に欄を作る.

8.2.5.2 アルゴリズム

記憶域割当てのアルゴリズムを作る前に，実行時に効果的な参照ができるように情報記憶の書式を考えておかなければならない．目的コードが実行される機械について，どのようなレジスタが使用できるか，記憶装置はどのようにして参照されるか，間接番地指定は可能か，インデックスやベース・レジスタは使えるかなど，幾つかの質問をしておく必要がある．記憶域の配置はこれらの問に対する答えによって定まる．（ここでは静的記憶域のみを考慮し，自動変数，被制御，および基底つき記憶域などのより複雑なクラスについては第3部にて取り扱う.）

ベースおよびインデックス・レジスタを持つ IBM 360 型計算機では，利用者の記憶域を，連続したブロックに配置し，各記憶域ブロックの開始点をレジスタに設定して指定することができる.



このようにするとすべての静的記憶域の割当てにはオフセットのみを計算すればよい．コード生成段階では(1)命令の中で使用される番地を生成するためにこれらのオフセットを使用して，(2)実行時にその記憶域が参照される前に，適当な記憶域へのポインタをレジスタにロードするコードを生成する.

記憶域割当て段階では，まず，記号表を走査し，静的なクラスの記憶域であることが指定されている欄に記憶場所を割当てる．この段階では零に初期化されたロケーション・カウンタに使用して，どれだけの記憶域を割当てたかを記憶している．記憶域割当て段階で走査中に静的変数を見つけると，次の4つのステップを実行する：

1. 必要な語境界の条件でロケーション・カウンタを整合させる.
2. ロケーション・カウンタの値を変数の番地部に割当てる.

3. その変数に必要な記憶域の長さを（属性を調べて）計算する．

4. この長さを加えてロケーション・カウンタを更新する．

この段階では，静的記憶域を要求しているすべての変数に相対番地を割当てたとき，行列に次の欄を作る：

STATIC	容量	

この欄を用いることにより，コード生成段階は必要な容量の記憶域を生成することができる．初期化を必要とする変数に対しては，記憶域割当て段階では行列に次の欄を生成する：

初期化	変数	

この欄の指示に基づきコード生成時に，活動ルーチンが記号表の中に保存しておいた初期値を適当な記憶場所に設定する．

記号表の走査が自動記憶域および被制御記憶域に対しても上と同様に行なわれ，各欄に相対番地が記入される．“自動変数”の欄および“被制御変数”の欄も行列の中に作られる．コード生成段階では命令の番地部を生成するのにこの相対番地を使用する．コンパイル時には自動変数または被制御変数に対する記憶域は生成されない．しかし，実行時にこの記憶域を割付けるコードが行列の自動変数であることを示す欄によって生成される．このようにして生成されたコードを実行すると自動変数の記憶域が割付けられる．

リテラル表も同様に走査され，各リテラルに番地が割当てられ，行列の欄

LIT	容量	

が作られる．コード生成段階では，静的記憶域にすべてのリテラルのための記憶域を生成し，リテラルの値で初期化する．

一時記憶域については，各文は前の文が使用した一時記憶域（中間的な行列の結果の領域）を再使用するので，上とは異なった処理を受ける．各文に必要となる一時記憶域の容量の計算が行なわれ，最も大きな領域を必要とする文の所要量がプログラム全体を通じての一時記憶の容量とされる．そして次の形の行列の欄が作られる：

AUTOMATIC	容　量	

この欄によってコード生成段階では適当な容量の記憶域を作るコードを生成する．一時記憶域は，このプログラムによってのみ参照され，このプログラムが実行される間でのみ必要となるので，自動記憶域である．

8.2.5.3 例

例題プログラムの記号／一時記憶域表，リテラル表，および行列の幾つかの欄は**図 8.23** に示されている．記号表の中のすべての変数は固定小数点 2 進数である．したがって，これらは各々 1 語に割当てられている．リテラルは 10 進数で，IBM 360 ではパックされた形式を用いて 2 数字毎に 1 バイトを割当てている．リテラル記憶域に対する行列の欄はこの領域の長さが 3 バイトであることを陽に示し，この領域がコード生成時にリテラル表の値を用いて初期化される必要のあることを陰に示している．

これからのことから，データ参照に関して種々の問題の発生することが予想できる．後の段階では参照される各変数に対して物理的な番地を生成しなければならない．静的記憶域に対しては少なくとも各変数の相対番地はわかっている．しかし，被制御記憶域についてはコンパイル時はおろかロード時においてすらこの記憶域が存在するかどうかわからないし，ましてその番地などはわからない．

データの型が混合している場合の処理にも注意しなければならない．リテラルは 10 進数で，記号は固定小数点 2 進数である．コード生成では行列の演算子だけではなく演算数のデータの型も調べる必要があり，それらの使用法によっては適当な変換ルーチンを作る必要がある．

8.2.6 コード生成段階

コード生成段階の目的は適当なコード（アセンブラ語または機械語）を生成することである．本節でも一般的な方法を用いてコード生成法および機械に関係する最適化法の問題点を明らかにして行く．多くのコンパイラは本節に述べる手法を使用している．この方法では，行列の中に現われる演算子を定義するコード生成則（マクロ定義）を用いてコードを生成する．また，記号表とリテ

記号表

	名前	基数	尺度	精度	記憶域のクラス	その他	番地
1	WCM	proc name					
2	RATE	BINARY	FIXED	31	STATIC		0
3	START	BINARY	FIXED	31	STATIC		4
4	FINISH	BINARY	FIXED	31	STATIC		8
5	COST	BINARY	FIXED	31	STATIC		12
	MTX5	BINARY	FIXED	31	AUTOMATIC		0
	MTX6	BINARY	FIXED	31	AUTOMATIC		4
	MTX7	BINARY	FIXED	31	AUTOMATIC		8
	MTX8	BINARY	FIXED	31	AUTOMATIC		12
	MTX9	BINARY	FIXED	31	AUTOMATIC		16
	MTX10	BINARY	FIXED	31	AUTOMATIC		20
	MTX11	BINARY	FIXED	31	AUTOMATIC		24

リテラル表

リテラル	基数	尺度	精度	その他	番地
31	DECIMAL	FIXED	2		
2	DECIMAL	FIXED	1		0
100	DECIMAL	FIXED	3		1

行列

演算子	演算数 1	演算数 2	連鎖
STATIC	16		
AUTOMATIC	28		
LITERALS	3		

図 8.23 記号表，リテラル表，行列——記憶域割当ての後

ラル表を参照して適切な番地指定を行ない，データの型を適当に変換するコードを生成する．

図8.12 の例の中央の列のコードは **図8.9** の単純な定義を用いて生成したものであった．**図8.12** の最後の列に示されたようなもっと効率の良いコードを生成するにはより柔軟な定義が必要である．これを実現するために，定義の中でマクロ擬似命令を使用する．

8.2.6.1 データ・ベース

行列——各欄は，原形コードに定義されている演算子を含む．コード生成段階では各欄を調べ，原型コードを探してコードを生成する．いい換えると，各欄の演算子は演算数を引数とするマクロの役割りを果している．

記号表，リテラル表——これらを用いて変数（行列の演算数）のデータの型おび番地を決定し，正しい番地を参照してコードを生成する．また型の混合した式が存在するときは，コード生成段階において適当な変換のためのコードが生成される．

コード生成則（マクロ定義）——行列のすべての演算子を定義している固定の表．第1部の**図8.9**は，演算子“−”,“＊”,“＋”,“＝”に対する単純なコード生成則を示している．ここでは，機械に関係する最適化が可能となるように，能力の大きいコード生成則用言語を用いて“＋”演算子を定義し直してみよう．コード生成則用言語として IBM 360 マクロ言語を使用することにする．実際の組込みにおいて設計者は自分用の言語を作り出せばよい．

8.2.6.2 アルゴリズム

本節ではコード生成処理をマクロ・プロセッサに使用されたのと類似した方法で組込む．行列の各欄の演算子部は“マクロ呼出し”として処理され，その欄の演算数は“マクロ引数”として使用される．第4章ではマクロ・プロセッサの問題を論じた．ここでは，機械に関係する最適化が可能となるよう，演算子を定義することに焦点を合わせよう．

8.2.6.2.1 簡単な機械に関係する最適化

図8.24 は原文 A＝B＋C＋D に対する行列を示している．中央の列は**図8.9**

行 列	"元のコード"	"改良されたコード"
+ B C	1) L 1,B	1) L 1,B
	2) A 1,C	2) A 1,C
	3) ST 1,M1	—
	4) L 1,M1	—
+ M1 D	5) A 1,D	3) A 1,D
	6) ST 1,M2	—
	7) L 1,M2	—
= A M2	8) ST 1,A	4) ST 1,A

図 8.24 文 A=B+C+D に対するコード

の"+"および"="に対する単純な定義を用いて生成されたコードを示す．3列目のようなより優れたコードを作る方法を考えてみよう．

2つの型の最適化が用いられている．第1，値がすでにレジスタに入っているときにはロードし直す必要はない．これによって L 1,M1 および L 1,M2 を消去できる．第2，行列生成の過程を注意深く考察してみれば，行列の一時的要素 M*i* は高々1回しか使用されていない．（このことは，行列の最適化が行なわれているときは正しくない．しかしそのときでもこの問題は解決できる．ここでは行列に対して中間的な最適化が行なわれていないものと仮定しておく．）したがって，レジスタに入れられた M*i* の値を直ちに使用するのであれば，この値を保存する必要はない．このことから命令 ST 1,M1 および ST 1,M2 が消去できる．

コードの最適化生成を行なうのには3つの方法がある：
(1) 行列の最適化のとき，(2) コード生成のとき，または(3) アセンブラ・コードが生成された後に．行列は，最適化されたときでも，原始プログラムの機械に独立な表現であることを目的としている．行列の中でレジスタ使用の最適化を行なうと行列は**機械に関係する**ものとなってしまう．たとえば IBM 1620 や 1401 のように，"レジスタ"のない機械のときにはこのような行列は何を意味することになるのか．形式の統一と実用上の両方の理由によって，行列は機械に独立のままにしておきたい．

コード生成後に最適化を行なうことは多くの欠点がある．たとえば，100 行の欄を持つ行列は 4000 枚ものアセンブラ・カードを生成する．行列のときはコアの中で操作ができたのに，アセンブラ・カードを出力したり再び読込んで不要な命令を消去することは時間のかかる処理である．最も良い方法はコード

を生成する**前に**不要な命令を取り除くことである(すなわち,コード選択時に).

8.2.6.2.2 機械に関係する簡単な最適化アルゴリズム

図 8.25は,“+” 演算に対して不要なロード(L)および格納(ST)命令を消去するアルゴリズムを示している.

このアルゴリズムには2つ注意せねばならない点がある.第1,演算数 **M*i*** がすでに(実行時に)レジスタに入っていれば,再びロードしなくてもよい.

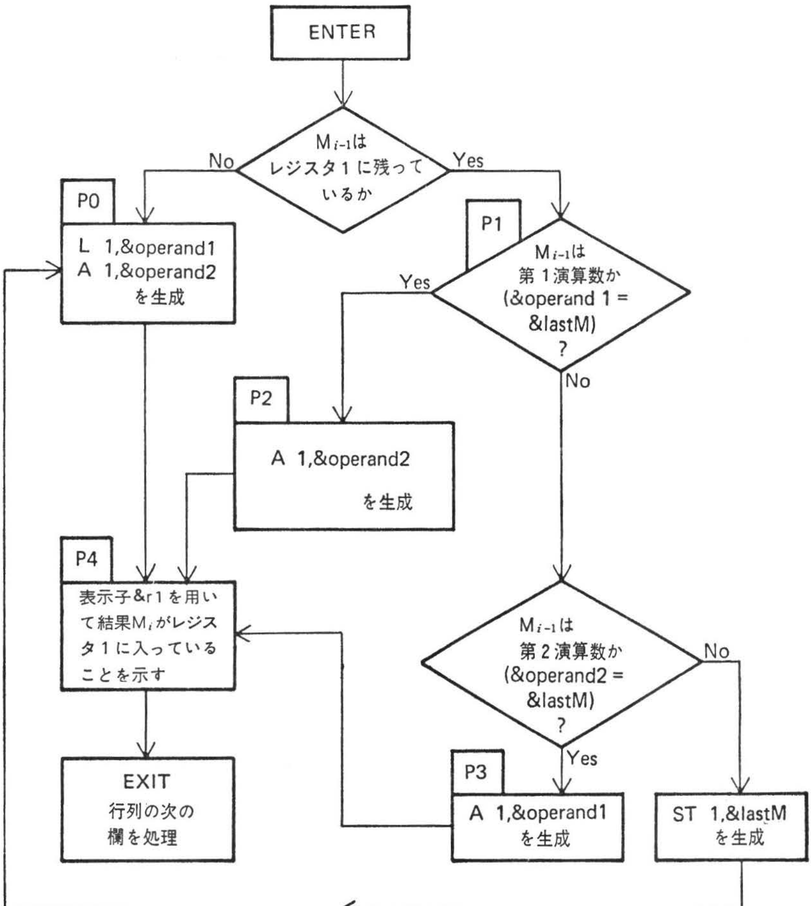


図 8.25 “+” 演算子に対するレジスタ使用の最適化を行なったコード生成言語

第2，行列の一時的要素を格納することは行列の**次の**欄のコード生成に任かせる．その理由は，前に使用された一時的要素が直ちに必要となるか否かは後の欄でのみ判定し得るからである．

各演算子のコード生成に対して別々のアセンブラ・サブルーチンを作らずに，原形コード言語を用いて適当なアルゴリズムを表現する．

“＋”演算子に対しては**図8.26**の原形コードが使用できる．簡単のためにこの原形においては IBM 360 型のマクロ言語の規約を用いている．

変数 &i, &r1, &M および &lastM はコード生成の最初で次のように初期化されていると仮定する（IBM 360 マクロ言語の記号による）．

```
          GBLA     &i
          GBLC     &r1 , &M , &lastM
&i        SETA     1
&r1       SETC     'null'
&M        SETC     'M1'
&lastM    SETC     'null'
```

“＋”の原形コードの中で使用されているマクロ NEXTM は次のように定義できる：

```
          MACRO
          NEXTM
&lastM    SETC     'M&i'     Mi-1 の値を設定する
&i        SETA     &i+1      i の値を設定する
&M        SETC     'M&i'     Mi の値を設定する
          MEND
```

“＋”の原形コードで用いられているラベル P0, P1 などは**図8.25**の流れ図の箱につけられたラベルに対応している．読者は流れ図と原形コードを比較して正しく処理が行なわれることを確認してみてほしい．また，上に示した原形コードを用いて前に出てきた行列を実際に処理してみることをおすすめする．

8.2.6.2.3　複数のレジスタを用いる最適化

条件つき原形コードは始めに述べた単純なものよりもっと効率の良いコード

&operand1	—	行列の欄の第1演算数
&operand2	—	行列の欄の第2演算数
&M	—	行列の現在処理中の欄 (M_i) に関連した一時記憶域
&lastM	—	行列の前に処理した欄(M_{i-1})に関連した一時記憶域
&r1	—	レジスタ1の内容を示す表示子 (すなわち、レジスタ1に何か残されているとしたらM_iとなる),
&i	—	行列の処理中の欄番号

	+に対するコード原形		意味
	AIF	('&lastM' EQ '&r1').P1	M_{i-1}がR1に入っていればP1に行く
.P0	L	1,&operand1	命令を生成
	A	1,&operand2	命令を生成
	AGO	.P4	EXIT手順に行く
.P1	AIF	('&operand1' EQ '&lastM').P2	&operand1=M_{i-1}ならばP2に行く
	AIF	('&operand2' EQ '&lastM').P3	&operand2=M_{i-1}ならばP3に行く
	ST	1,&lastM	命令を生成
	AGO	.P0	P0に行き続行
.P2	A	1,&operand2	命令を生成
	AGO	.P4	EXIT手順に行く
.P3	A	1,&operand1	命令を生成
.P4	ANOP		EXIT手順の開始
&r1	SETC	'M&i'	&r1をM_iに設定する
	NEXTM		&i, &M, および&lastMを更新する
	MEND		次の欄へ行く

図 8.26 ある程度の最適化を行なった+マクロに対するコード原形

を生成する．しかしそれに伴う欠点も存在する．

A=B＊C+E＊F；

に対して**図 8.27** の行列を考えてみよう．

前節の“+”と同様にして“＊”に対する定義を仮定すると，中央の列に示したような“最適化された”コードが生成される．中央のコードでは不要なロードや格納が消去されているけれども，最後の列のコードの方がもっと“能率が良い”．

行列				コード			改良コード	
1)	*	B	C	1)	L	1,B	1) L	1,B
2)	*	E	F	2)	M	0,C	2) M	0,C
3)	+	M1	M2	3)	ST	1,M1	3) L	3,E
4)	=	A	M3	4)	L	1,E	4) M	2,F
				5)	M	0,F	5) AR	1,3
				6)	A	1,M1	6) ST	1,A
				7)	ST	1,A		

図 8.27 A=B＊C+E＊F に対するコード最適化

複数のロードや格納を消去しても得られる改善度は 50% に達するということはない．しかし，1つの命令を消去し，加算命令をレジスタ加算命令に置換えるだけで 15% も能率が良くなるということもある．

この違いは，IBM 360 は算術演算を実行できる2個以上のレジスタを持っていることにある．16 個のレジスタの幾つかはベースおよびインデックス・レジスタとして確保しておかなければならないが，算術演算用として少なくとも2〜3対のレジスタを用意することはできよう．

図 8.28 は“＋”演算子に対するコード生成処理のアルゴリズムを示している．この操作の基本は，アセンブラのベース表の機能に類似したレジスタ表示子の概念である．算術計算に対して使用される各レジスタに対してそれぞれ1つの表示子が用意され，(1)レジスタは中間結果を格納していないか，または (2) レジスタに M*i* が置かれているかを示している．

記号 &R1 および &R2 は，“レジスタ変数”であり，値として0から15をとる．行列の行 *i* の結果がレジスタ *j* に残されているとき，&r*j*=M*i* および &R1=*j* と設定する．たとえば，命令 AR &R1,&R2 は，&R1 が値3を，&R2 が値1を持つとき，AR 3,1 を生成する．

命令 ST &R1,&r&R1 は，レジスタを対応した行列の一時的要素に格納する命令として生成される．たとえば，&R1 が値3を，レジスタ3が行列の行39の中間結果を納めているとすると，生成される命令は；

ST 3,&r3 これは次のようになる ST 3,M39

このことは値が実際に計算された後に M*i* への格納命令が無数に生成されることもあり得ることを意味している．

8.2.6.2.4 実際上の問題点

本節では，コード生成段階で処理されなければならない幾つかの問題を論じ，簡単のために，IBM 360 マクロ・プロセッサ（第4章に述べている）の形式でその機構を述べてきた．種々の方法があるがコード生成はコンパイラの最も複雑な段階である．

コンパイラ作成者にとって最も難しい仕事は，.procbgn や .rtn その他の非算術演算子をも含めて，すべての行列の演算子に対して効率の良い原形コー

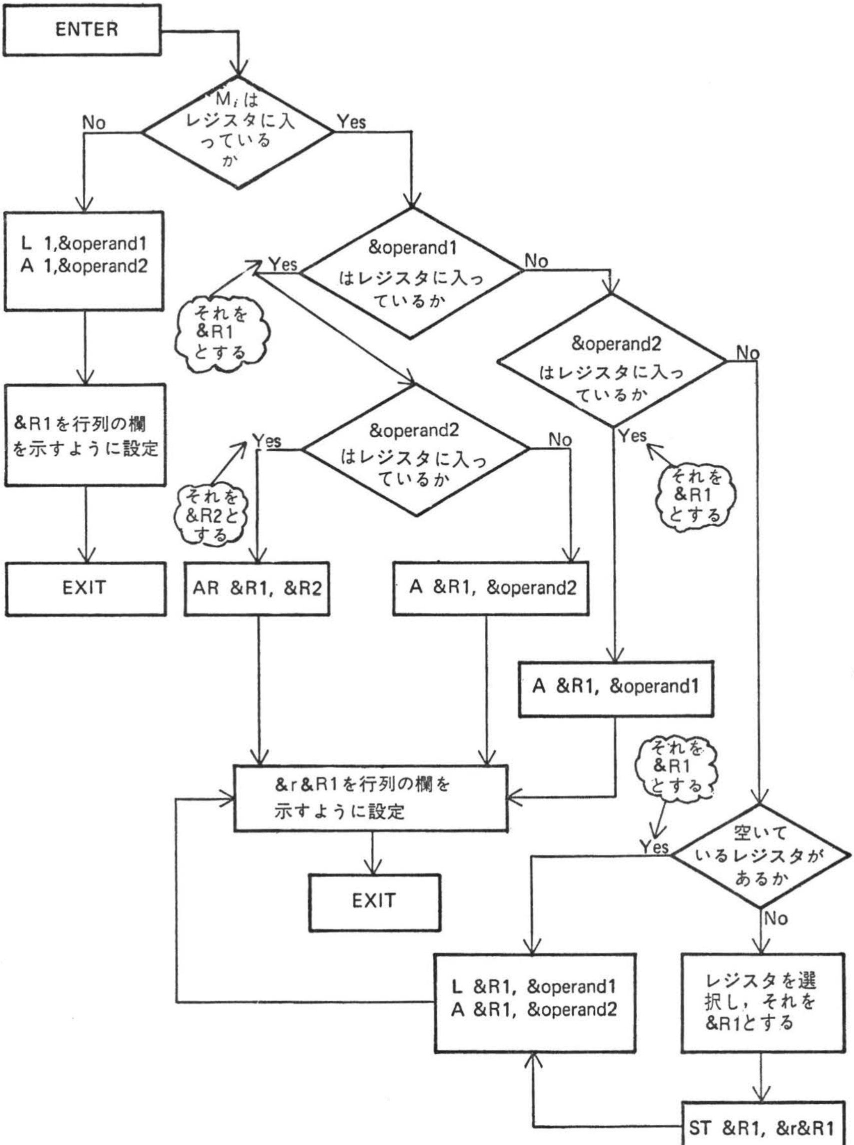


図 8.28 “＋”演算子に対するコード生成の処理——改良型

ドを定義しなければならないことである．さらに，行列の中の“＋AB”といった算術演算子に対してAとBが同じデータの型（たとえば，2進と10進）でないときや，“＋”演算子が適用できないとき（たとえば，文字列やビット列）のときにはデータ変換を行なわねばならないことである．

これらの問題は十分な定義能力を持つ条件つき原形コードの定義によって解決できる．しかし，コンパイルの速度ならびに効率，さらにには能力と柔軟性を得るために，一般的なマクロ・プロセッサを使用せず，特別のコード生成処理と原形記述言語を使用することが多い．

コード生成段階の処理の程度によっては，残りの段階の処理がきわめて単純になることがある．たとえば，コード生成段階でアセンブラの原始コードが生成されるとき，必要とされる機械語の目的デックを生成するのに1パスまたは2パス・アセンブラを使用する必要がある．完全な形でコンパイラの記号表が与えられるときは1パス・アセンブラが適当である．DSやDC文が作られるときは完全な2パス・アセンブラが必要となる．AED 360や初期の7090 FORTRANなどの幾つかのコンパイラは，簡単のために，実際にアセンブラの原始コードを生成し，標準システムのアセンブラによって翻訳処理を完結させている．7090 MADなど多くの高速高性能のコンパイラは，コード生成段階の出力として直接完全な機械語を生成し，アセンブル段階をなくしてしまおうとしている．

8.2.7 アセンブル段階

アセンブル段階の仕事は，コード生成段階でどこまで処理がすんでいるかに依存する．もしコード生成段階で多くの処理が行なわれていれば，アセンブル段階は単純で，目的プログラム中のラベル参照を解決し，目的デックの形式を整え，ローダに対して適当な情報を用意すればよい（たとえば，直接結合ローディング方式では TXT, RLD, ESD カードを用意する．第5章参照）．

極端な場合でコード生成段階が単に記号表現による機械命令とラベルのみを生成しているときには，アセンブル段階は複雑で，(1)ラベル参照を解決し，(2)番地を計算し，(3)2進数表現の機械語を生成し，(4)記憶域を生成し，リテラルを変換しなければならない．

8.2.7.1 データ・ベース

記号表——簡単な方のアセンブル段階ではこのデータ・ベースを用いてすべてのラベルの値を記入する.

リテラル表——複雑な方のアセンブル段階ではすべてのリテラルの番地を初期化し，適当な TXT カードにこのリテラルを記入する.

目的コード——コード生成の出力.

8.2.7.2 アルゴリズム

簡単な方のアセンブル段階では目的コードを走査して，すべてのラベル参照を解決し，TXT カードを生成する.次に記号表を走査して ESD カードを作る. RLD カードは，目的コード，ESD カード，および記号表を用いて作られる.

もっと複雑なアセンブル段階は第3章で述べたのと同様のアセンブラである．実際の組込みにおいてはルーチンの機能やデータ・ベースの一部，たとえば,記号表などはコンパイラの他の段階と重複するので省略することができる.

8.2.8 コンパイラのパス

コンパイラの論理的な7つの段階の代りに，データ・ベースを走査する物理的なパスの数Nによってコンパイラを考えることもできる.

図 8.29 はコンパイラの流れ図の概要であり，パスを示している.

パス1は単語段階に対応する．このパスでは原始プログラムを走査し，記号表，リテラル表，および不偏記号表を作る.

パス2は構文段階と解釈段階に対応する．パス2では不偏記号表を走査し，行列を生成し，記号表に記号に関する情報を記入する．パス1と2は，必要なときに原始プログラムを解析し直接スタックにトークンを入れる活動ルーチンとして単語段階とみなすと，1つのパスにまとめることができる.

パス3からパス N−3 は最適化段階に対応する．各型の最適化には行列を数回走査することが必要である．特定のコンパイラに組込まれている最適化の技法は様々であり， PL/I のようにプログラマによって制御できるものもある.

パス N−2 は記憶域割当て段階である．このパスでは記号表とリテラル表が走査され，プログラム自身には関係がない.

パス N−1 はコード生成段階に対応する．このパスでは行列を走査し，目的デックの第1版を生成する.

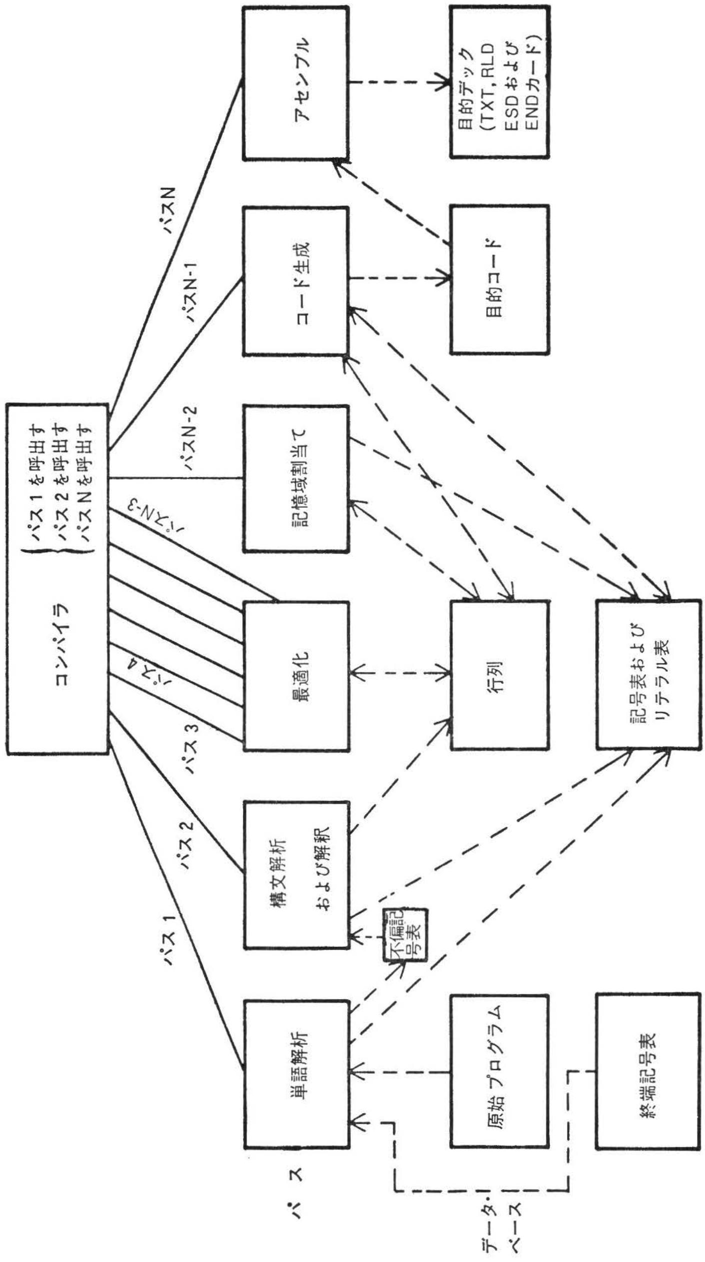


図 8.29 コンパイラのパス

パスNはアセンブル段階に対応する．このパスでは記号番地指定を処理し，ローダのために情報を作る．パスNはおおよそのところ第3章で述べたアセンブラのパス2に対応する・マクロ機能や他の高度の機能がコンパイラに追加されると，さらに余分のパスが必要となる．

8.2.9 予　告

第3部では，このモデルを用いて複数のデータの型，複数の記憶域のクラス，およびその他の機能を取り扱う．また，データ・ベースの情報を拡大し，活動ルーチンの機能を増大させる．

第 3 部

第3部では前のモデルに対して次にあげる高度の機能を組込む：

1. データ構造
2. 回帰性
3. 記憶域のクラス
4. ブロック構造
5. 条件
6. 非局所 GO TO
7. ポインタ

8.3 データ構造

図8.30では構造Aを**静的**と宣言している．この構造は6つの単純変数を持っている．Cは固定小数点2進数，Eは文字列，Fはビット列，Gは固定小数点2進数，などとして宣言されている．この構造はAとBが節でCが単純変数である樹と考えることもできる．Cを参照するにはCへの経路名 A.B.C を用いて行なわれる．この構造は6つの記憶場所から成るn項子とも考えることができる．各記憶場所は C,E,F,G,J およびKによって指定されるデータを納めている．構造の全体Aはすべての記号場所を指定する．この構造は100個の欄を持つ配列として宣言されているので，100個の樹またはn項子が存在する．プ

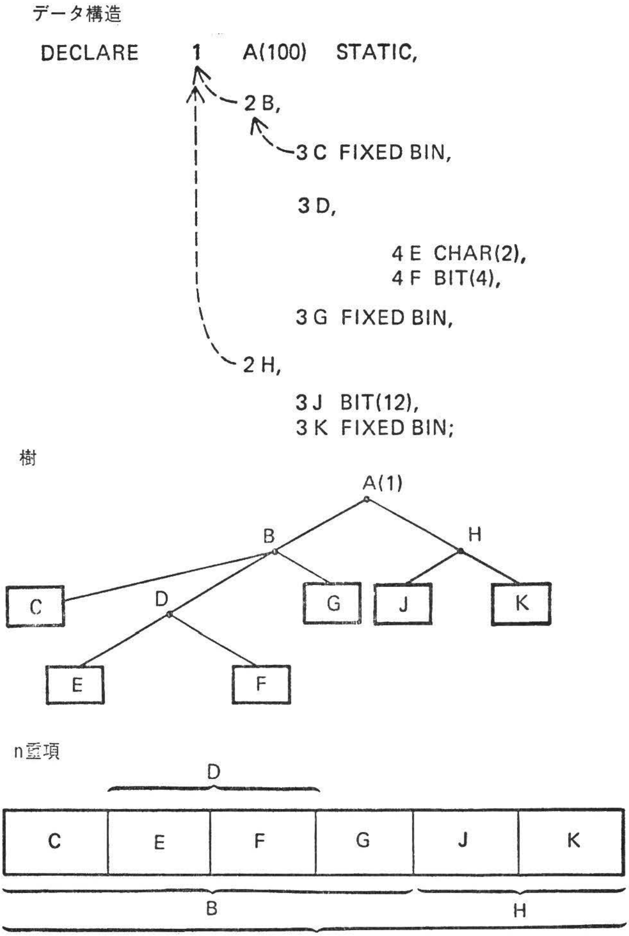


図 8.30 データ構造の表現

ログラムの中でこの構造の宣言の後では A(2).B.D.E='NO' などといった文を書くことができる．なお PL/I で添字は，たとえば，A.B.D.E(2) のように後に置いてもよいことになっている．

8.3.1 問題の設定

データ構造とそれを参照している文を処理するには，コンパイラは次の4つの仕事を行なわなければならない：

1. 構造の宣言に含まれる情報を記号表に格納する． この情報は，記憶域割当て段階で記憶域を割当てるのに使用され，コード生成段階で適当な参照を行なったり型の変換を行なうコードを生成するために使用される．コンパイラは構造の中のすべての記号（たとえば A, B, C, D, E, …, K）に対して次の項目を記入する：

名前	基数	尺度	精度	記憶域のクラス	配列の限界	ブロック情報	構造名，節，単純変数の情報	接続情報	初期値	長さ	番地

すべての名前に対する欄

単語段階ではこの欄に名前を記入する．解釈段階で基数，尺度，精度，配列の限界，ブロック情報，構造であるかどうかの表示，節／単純変数の表示，接続情報，および（存在すれば）初期値を格納する．接続情報は部分構造の"親"を指す後向けポインタから成る（**図 8.30** 点線を参照）．これらのポインタを用いることによってプログラム中で構造体の変数への参照が完全に記述されていなくても処理が可能であり，その参照を不偏記号で置換えることができる．記憶域割当て段階では長さと番地を格納する．長さはこの要素およびすべての部分構造，すなわちそれに含まれる要素，に必要な記憶域の容量である．

2. すべての参照を不偏記号に変換する． 解釈段階では構造への参照に対して不偏記号を作る．単語段階では構造の各記号に対して記号表の中に欄が作られている．A.B.D.E という参照は記号表の中のEを指す不偏記号によって置換えられる．正しいEを見つけるには，Eという名前の記号を探し，ポインタを逆にたどってその親がDであり，Dの親がBであり，最後にBの親がAであることを確かめる．同様にして，A.E という不完全な参照が行なわれると正しいEを指す不偏記号で置換えられる．またA.B という参照は正しいBを指す不偏記号で置換えられる．この参照は単純変数 C, E, F, G を示している．

3. 記憶域を割当てる． 記憶域割当て段階では記号表に長さと番地を記入しなければならない．また，適当な欄を行列の中に作って記憶域が割当てられ初期化されるようにする．構造の底を指すベース・レジスタが使用できるよう

な IBM 360 型の計算機の場合に，**図 8.30** の例の構造に対しては記憶域を順番に割当てることができる．例の場合には**図 8.31** のようになる．

記号表にはすべての欄の長さが記入される．単純変数の場合の欄の長さはデータ要素の長さである(たとえば，**図8.30**のGの長さは4バイトである)．部分構造の場合の長さは部分構造のすべての単純変数の長さの和に等しい(たとえば，**図8.30**のDの長さは3バイトである)．長さの部分はコード生成において適当なコードを生成するのに使用される(たとえばA．B(1)=A．B(3) に対しては部分構造を移すコードが作られる)． ステップ1から3を例に適用すると**図 8.32** に示した記号表が作られる．読者は自分で長さと番地を計算してみてほしい．

4. コードと適当な番地を生成する． コード生成段階では適当な記憶域を割付けてそれを初期化するか，または実行時に記憶域を割付けるコードを生成するかしなければならない．データ構造を使用するときには実際の番地の生成を行なわねばならず（記号番地を生成するのに比べて）一層複雑となる．たと

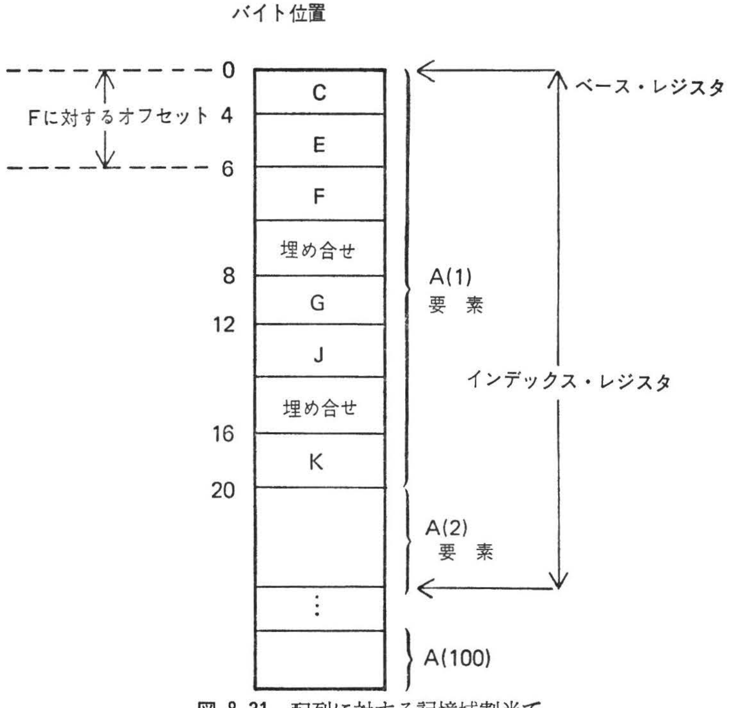


図 8.31 配列に対する記憶域割当て

指標	名前	基数	尺度	精度	記憶域のクラス	配列の限界	ブロック情報	構造情報	接続(親の長さ)	初期値	長さ(バイト)	番地
1	A				STATIC	0,100		構造	0	0	20	0
2	B							部分構造	1	0	11	0
3	C	BINARY	FIXED	31				単純変数	2	0	4	0
4	D							部分構造	2	0	3	4
5	E	CHAR	FIXED	2(文字)				単純変数	4	0	2	4
6	F	BIT	FIXED	4				単純変数	4	0	1	6
7	G	BINARY	FIXED	31				単純変数	2	0	4	8
8	H							部分構造	1	0	6	12
9	J	BIT	FIXED	12				単純変数	8	0	2	12
10	K	BINARY	FIXED	31				単純変数	8	0	4	16

図 8.32 図8.30の例の記号表の欄

えばコンパイラは命令 L 1,A.B.C(1) を生成することができず，L 1,4(8,9) を生成しなければならない．ただし，4(8,9) は実際の番地を示す‘A.B.D.E(5) をロードする’というような参照に対してコード生成段階では次の順にコードを生成する：

a. 構造Aの底をベース・レジスタ(たとえばレジスタ 9)に設定するコード．

b. インデックス・レジスタに (5−1)×(構造の長さ) に等しい指標の値を設定するコード．

c. 記号表の中のEに対する番地を値をオフセットに入れ，ステップaのベース・レジスタと，ステップbのインデックス・レジスタを指定したロード命令 (例えば，L 1,4(8,9))．

上の3つのステップは A(5).B.D.E の番地を形成する．なお，ステップbではインデックスの値をコンパイル時にあらかじめ計算せずに，実行時に計算するコードを生成している．これは，もっと一般的な，添字が，たとえば A.B.D.E(DX+1)のように，実行時に始めて計算できる変数や式の場合に適用することを意識している．

8.3.2 組込み

ここではステップ1から4までの組込みの詳細を手短かに述べる.

構文段階および解釈段階——まず,宣言文の構文の正確な記述を行なう(たとえば,BNFにより).適当と考えられる活動ルーチンを定義し(たとえば,ENTERは配列の限界, 底, 精度を記入するなどと), 変形則を書く. 解釈段階で宣言文を処理するときには記号表に情報を記入することに留意. 解釈段階で添字つき変数を参照する算術文を処理するときには, 行列の欄を作って, 番地を計算するコードを生成するようにする. たとえば, A.B.C(I)=3 に対しては次の行列を作る:

演算子	演算数1	演算数2
.SS	A.B.G	I
=	M1	3

コード生成段階では, この欄によってインデックスの値を計算し適当な番地を形成するコードを生成する.

単語段階および記憶域割当て段階——上述のとおり.

コード生成段階とアセンブル段階——宣言文に対応するコードは存在しないが,データ構造の中で定義されている要素を参照するためのコードは存在する. 文 A=B は行列の欄 | = | A | B | に変換される. 単純変数に対しては, 記憶域割当て段階で記号表の欄の番地部に領域の先頭に対する相対番地を記入する. この単純な算術文に対しては, コード生成段階でAとBの番地を形成せねばならない. これらの番地は3つの部分, オフセット, ベース・レジスタ, およびインデックス・レジスタ(単純変数に対しては使用されない)より成る.

オフセットは記号表の中の記号の欄を探して指定することができる. ベース・レジスタ番号は任意に選ぶことができるが, コード生成段階ではベース・レジスタに静的記憶域の開始番地を設定するコードを生成しなければならない.

添字つきのデータ構造に対する参照の場合には,番地形式がやや複雑となる. 文 A.B.C.G(I)=3 が与えられたとしよう. この文に対する行列は前述のとおりである. マクロ“.SS”は, A.B.G(I) の**番地**を計算し, その番地をこの行列の欄に関連した一時記憶域に格納するコードを生成する. 実行時にこの番地を計算するために, コンパイラは構造Aの長さとAの中でのGの位置の情報

を与えなければならない．

繰り返すと，番地は，オフセットとベース・レジスタの内容とインデックス・レジスタの内容の，**3つ**の要素から構成される．オフセットはコンパイル時にわかり，その時点で生成される．しかし，ベースとインデックス・レジスタの内容は実行時に計算されなければならない． IBM 360 型の計算機ではベース・レジスタに記憶域の開始番地を設定する命令を生成することができる．また，配列の開始番地から特定の要素への距離の値を計算する適当な演算命令を生成することができ(たとえば，(I−1)×(要素Aの長さ))，この値をインデックス・レジスタに設定するコードが作られる．マクロ“＝”は2つの引数としてA.B.G(I) の番地とリテラル3の番地を持つ．これは第2部の“＝”の定義と異なっている．第2部でのこれらの引数は演算数の**値**を納めている番地であった．

8.4 回帰性， CALL 文および RETURN 文

回帰手続き（自分自身を呼出す手続き）をコンパイルするときの問題点を例を用いて説明しよう．例は階乗を計算する PL/I ルーチンである（他の例は第4.3.3 節を参照）．ここではすべての記憶域は静的であると仮定する．

N! を次のようにして計算してみよう：

$$N!=\begin{cases} N*(N-1)! & N>0 \text{ のとき} \\ 1 & N=0 \text{ のとき} \end{cases}$$

図 8.33は 3! のアルゴリズムを示している．これは正しくない．これを用いて回帰手続きの問題点を説明する．アルゴリズムは単に N!＝N*(N−1)! であり，したがって， 3!＝3＊2!＝3＊2＊1!＝3＊2＊1＊0!＝3＊2＊1＊1 である．すなわち，Nから1を減じて再び階乗を呼出せばよい．N＝0 となれば復帰する．

図 8.33 の手続きは 3! を計算し，結果を ANS に入れる．N, ANS, TMP に対して，第2部で述べられたようにしてプログラムの終りに記憶域が割当てられている．**図 8.34** は呼出し時のこれらの変数の値を示している． TMP は復帰のときにつね同じ値をとるため，プログラムは正しい答えを与えない．

```
FACT:   PROCEDURE;
        DECLARE N FIXED BIN STATIC INITIAL (3);
        DECLARE (ANS,TMP) FIXED BIN STATIC;
        IF N = 0
             THEN ANS = 1 ;
             ELSE DO;
                  TMP = N;
                  N = N-1;
                  CALL FACT;
                  ANS = TMP * ANS;
                  END ;
        RETURN;
END;
```

翻訳結果

```
        START
        SAVE                 ←
        L       4,N
        C       4,=F'0'          IF N = 0
        BNE     LA
L       L       4,=F'1'
        ST      4,ANS            THEN ANS = 1
        B       LB
LA      L       4,N
        ST      4,TMP            TMP = N
        L       4,N
        S       4,=F'1'          N = N-1
        ST      4,N
        CALL    FACT         ←
        L       5,TMP
        M       4,ANS            ANS = TMP * ANS
        ST      5,ANS
LB      RETURN               ←
N       DC      F'3'
ANS     DS      F
TMP     DS      F
        END
```

図 8.33 誤った回帰的プログラム

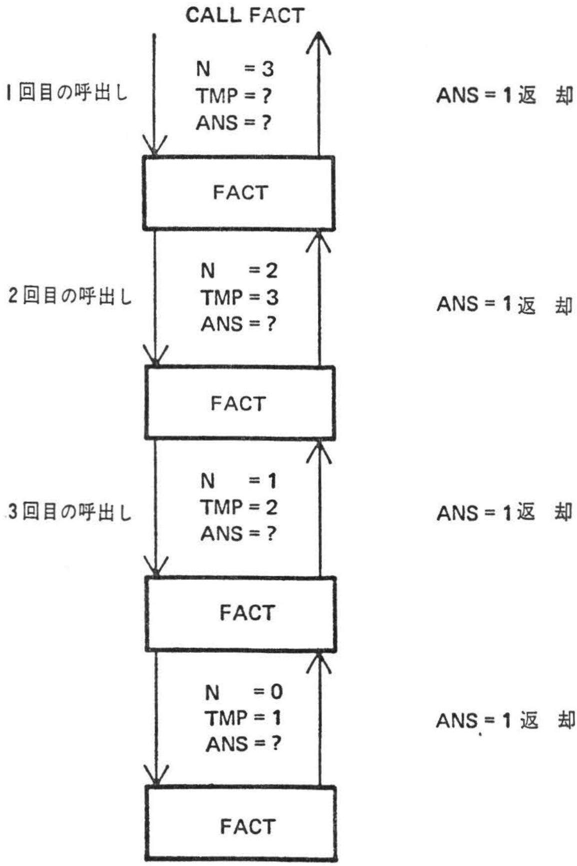
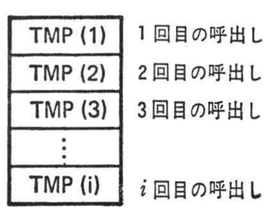


図 8.34　N! の計算――誤ったアルゴリズム

読者は実際にこの計算を追ってみてほしい．そうすればすべての復帰で TMP=1，したがって ANS=1 であることがわかるであろう．これを解決する方法の1つは TMP を配列とし TMP(I) とにいうように添字 I を用いること



である．呼出しの度に I=I+1 とし，新しい TMP の領域を確保する．復帰するときには I=I−1 とし，前の呼出しに対応した TMP を復元する．

このような変数の格納法は漠然とスタックと呼ばれている．I=I+1 とすること(新しい TMP を作る)は**プッシュ・ダウン**に対応し，復帰 (I=I−1) は**ポップ・アップ**に対応する．次節のデータ記憶域の議論で明らかになるように，自動記憶域と呼ばれる記憶域のクラスを使用することができる．自動記憶域は呼出しと復帰の際に自動的にスタックをプッシュしたりポップしたりする．

さて例題の問題点はまだ片づいていない．このプログラムは多分主プログラムから呼出されている．主プログラムはサブルーチンからわかる場所に復帰番地を残している．階乗プログラムが自分自身を呼出すと，同じ場所にその復帰番地を残すことになり，主プログラムへの復帰番地を壊してしまう．FACT が再び自分自身を呼出すと，前者の復帰番地は失われ，新しい復帰番地によって置換えられる．したがって，復帰したとき，最後の呼出しに対応した番地にしか復帰できない．上の例では，復帰番地はレジスタ 14 に残され，RETURN 文はレジスタ 14 の内容に対する分岐命令に翻訳される．また，CALL 文は，次のように翻訳される：

```
L        15,FACT
BALR     14,15
```

最初の呼出しにおいてレジスタ 14 は呼出し者の復帰番地を納めている．2回目から最後の呼出しまではFACT の中の復帰番地を納める．いかにしてFACT から外へ出るかが問題である．その答えはレジスタ 14 の内容もスタックに保存しなければならないということである．すなわち，一般には呼出しに関連した復帰番地を保存しなければならない．

多くの言語ではスタックの処理はコンパイラによって自動的に行なわれる．スタックの一部は復帰番地として使用され，他の部分は各呼出しに伴って変化する自動記憶域として使用される．また1つの手続きに入ったときすべてのレジスタを保存した方が効率が良い．このようにして，IBM 360 では次のスタック・フレームを用いている (図 8.35)．

他のプログラムを呼出すと，機械の状態と呼出しを行なった手続きの変数を納めるために新しいスタック・フレームを作ってスタックのプッシュ・ダウン

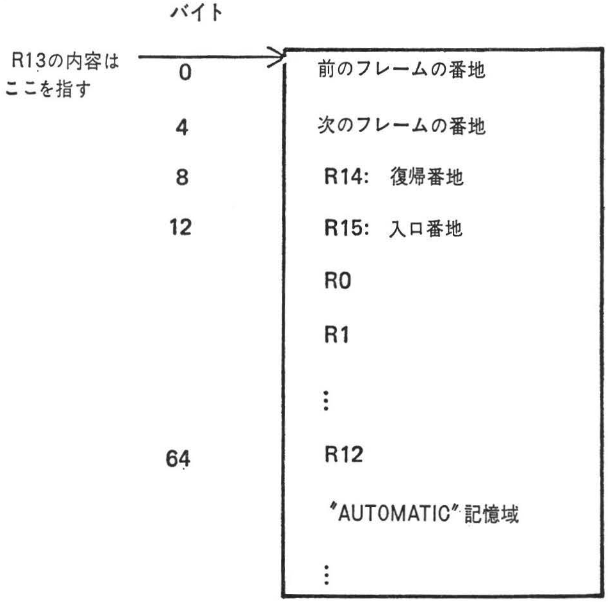


図 8.35 スタック・フレーム

を行なう．RETURN 文を実行すると，呼出し前の機械の状態を復元してスタック・フレームのポップ・アップが行なわれる．したがって，呼出された手続きに入るときは，機械の状態を保存するために SAVE 文を実行しなければならない．RETURN 文は機械の状態を復元する．CALL 文は新しいスタック・フレームを作る．

例の場合に使用できる単純な，SAVE, CALL および RETURN の手続きは**図 8.36** に示されている．これはよく理解しておかなければならない大切な事項である．読者は図の手順を追ってみてほしい．

回帰性を処理するとき，コンパイラはスタックを処理するために CALL, SAVE および RETURN 手続きを生成しなければならない．なお，これらの手続きがシステムで標準化されていれば，PL/I プログラムは FORTRAN プログラムを呼出すこともできるし，FORTRAN プログラムはアセンブラ語プログラムを呼出すこともできる．しかし多くのシステムでは異なったコンパイラが生成した目的プログラム同志を接続することはできない．

```
SAVE    ~   STM   14,12,8(13)      すべてのレジスタの内容をスタック・フレームに保存する
            LR    2,13        ←  コンパイラはここに   } 次のフレームの開始点
            A     2,=F'           フレームの長さを入   } を計算する
                                  れる（すなわち，68＋
                                  自動記憶域の長さ）
            ST    2,4(13)          このフレームのバイト4に次のフレームの
                                   番地を格納
            ST    13,0(2)          次のフレームのバイト0にこのフレームの
                                   番地を格納
CALL    ~   L     13,4(13)         次のフレームの番地をロード
            L     15,PROC_ENTRY    呼出された手続きの番地をロード
            BALR  14,15            レジスタ14に復帰番地を入れて呼び
                                   出したい手続きに分岐する
RETURN  ~   LM    14,12,8(13)      フレームからレジスタの内容を復元する
            L     13,0 (13)        このフレームのバイト0から旧フレームの
                                   番地をロードする
            BR    14               レジスタ14に入っている番地に分岐する
```

図 8.36　簡単な呼出し手順

SAVE, CALL および RETURN 手順は機械によって異なっている．ダイナミック・リンキング・ローダを述べるときに，HIS 645 に組込まれた MULTICS システムに対する呼出し手順を論じよう．

8.5　記憶域のクラス——使用法

今日の計算機利用者は FORTRAN, ALGOL および COBOL により用意されたものよりももっと柔軟な型の記憶域を要求し始めている．特に，必要に応じて記憶域を割付けたり解放したりしたいと望むであろう． PL/I はあまり良くない言語であるという批判は構文上のあいまいさに起因するようであるが，記憶域の割付けに関しては， PL/I の能力は利用者の要求を十分に満たすものと一般に認められている．

本節では種々の記憶域割付けとその使用法を論じる．次節では，それらの組込みを示す． PL/I が提供している記憶域の種類は一般性があるのでこれを選んで説明する． ALGOL 68 や新しい言語も同様な機能を用意している．

次の型の記憶域を論じることにする：

1.　内部静的記憶域

2. 外部静的記憶域
3. 自動記憶域
4. 内部被制御記憶域
5. 外部被制御記憶域
6. 基底つき記憶域

8.5.1 静的記憶域

内部静的記憶域はプログラムに局所的なものであり，実行に先立って割付けられる．すなわち，実行時にこの記憶域は存在し，使用されてもされなくても領域がつねに確保されている．また変数はすべての呼出しの間，元のままの値を保つ．

内部静的記憶域は手続きを**純粋でなく**，すなわち，**再入可能でなく**する．**純手続き**，すなわち，**再入可能手続き**は自分自身を変更しないし，変更されるデータや番地を持たない．純手続きについては第9章で多重処理について述べるときに論じる．

外部静的記憶域はつねに存在する内部静的記憶域と同様であり，実行の前に割付けられ，使用の有無に拘らず存在する．しかしこの記憶域は他の手続きによって参照することができる．これは FORTRAN の COMMON に相当する．

静的記憶域を使用する主な理由は2つある．第1，すべての呼出しに対して変化せず元のままの値を保つ記憶域のため．第2，静的記憶域を参照するために生成されるコードは最も単純であり，通常最も効率が良い．プログラムで静的記憶域が必要となる良い例は乱数発生であろう．通常，各呼出しに対して，プログラムは最後に生成された数を次の“乱数”を作る係数として用いる．したがって最後に作られた数は各呼出しを通じて保存されていなければならない．

8.5.2 自動記憶域

自動記憶域は局所的であるという意味で内部的である．そしてこれを定義している手続きの中でのみ参照することができる．記憶域は手続きが実行される

とき割付けられる．領域は手続きに入るときに割付けられ，手続きから出るときに解放される（これはスタックに相当する）．外部自動記憶域というものはない．この理由は，存在しないものを参照するということは無意味だからである．

自動記憶域の主な使用法は回帰呼出しに関してである．この場合回帰的な変数には各呼出しに対する値を持たせたい．したがって，自動記憶域として定義された変数に対しては，各呼出しごとに別の領域が存在する．回帰的な手続きからの復帰はこれらの変数を前に持っていた値に復元する．自動記憶域の他の使用法は記憶装置利用の効率を向上させるためである． FORTRAN プログラマを悩ませた問題でこれを説明しよう．サブルーチンの中では， FORTRAN プログラマはつねに，配列の大きさを呼出しプログラムが使用し得る最大値に宣言しておかなければならなかった．自動記憶域を使用すると配列は手続きを呼出すときに割付けられる．呼出しに際して配列の大きさは，次元をその手続きの引数として渡すことにより決定することができる．このようにして配列に必要な記憶域だけが割付けられる．

8.5.3　内部被制御記憶域

内部被制御記憶域を用いると利用者はプログラムの中で作業域を割付けることができる．記憶域が被制御と宣言されているとき，コンパイル時にも，それが宣言されている手続きが呼出されたときにもこの記憶域は存在せず，ALLOCATE 文を実行することによって割付けられる．FREE 文が実行されると被制御記憶域は消滅する．ALLOCATE 文が実行される度に変数の新しい領域が作られ，FREE 文が実行されると最も新しい変数の領域が解放される．したがって，2回目の ALLOCATE 文が FREE 文より前に実行されると，存在している記憶域はスタックの中でプッシュ・ダウンされる．FREE 文はスタックの先頭の記憶域を解放し，その前の領域をポップ・アップする．内部被制御記憶域は，それを定義している手続きによってのみ参照することができる．

8.5.4　外部被制御記憶域

外部被制御記憶域は，他の手続きによって参照したり解放したりできるという点で内部被制御記憶域と異なっている．外部被制御記憶域は柔軟であるが故

に，利用者はシステム・プログラマと同様に，データ・ベースに鍵を掛けたり外したりしなければならない．すなわち，外部被制御記憶域は1つの手続きで割付けられ，他の手続きによって解放され，3番目の手続きで参照しようとしたら存在しなかったなどというようなことも起こり得る．

8.5.5 基底つき記憶域

基底つき記憶域は ALLOCATE 文および FREE 文を用いて作ったり解放したりできるという点で被制御記憶域に類似している．被制御記憶域と異なるのは，記憶域のスタックへのプッシュやポップを伴わないということである．ALLOCATE 文を実行すると記憶域が作られる．基底つき記憶域を参照するために，その記憶域へのポインタが ALLOCATE 文によって設定される．基底つき記憶域を参照したり，解放したりするためにポインタを用いてその領域を指定する．

8.6 組　込　み

これらの記憶域に対してコンパイラは次の仕事を行なわねばならない：

1. そのクラスのすべての変数に番地を割付ける．
2. 記憶域が割付けられるように用意する．
3. 正しく参照が行なわれるように用意する．

8.6.1 静的記憶域

プログラムが静的と宣言した記憶域は，コンパイル時にコンパイラによって割当てられる．コンパイラは，ローダが実行中にこの領域を確保するための情報を出力する．通常，静的記憶域は命令と同じようにテキスト・カードの中に含められる．

外部静的と宣言された記憶域は同じくコンパイル時に割付けられるが，手続きセグメントとは別のセグメントに置かれる．外部記憶域の中の変数はすべて ESD カードに記録される．ここで読者はローダの節を復習して，直接結合ローディング方式で使用される TXT カード，ESD カード，および RLD カードの目的を思い出してほしい．別のセグメントとして外部記憶域を割付ける方法

はローディングの方式に依存するが，一般にローダはこのジョブのプログラムと外部静的セグメントの両方をロードする．ローダは２つの外部セグメントが同じ変数を含んでいないかを調べ，２重に記憶域を割付けないようにする．たとえば，P1 でAとBが，また，P2 でBが外部静的であると宣言されているとき，これら２つのプログラムをコンパイルすると，本体のほかに，各々に変数Bを含む２つの外部セグメントが作り出される．しかし，ローダはAとBに各々１つの記憶場所だけを作る．外部静的領域へのすべての参照は外部静的変数を指す“**トランスファ・ベクトル**”（一連のポインタ）を通じて間接的に行なわれる．このトランスファ・ベクトルはプログラムの内部静的領域に置かれ，ローダによってすべての外部静的変数の番地が記入される．外部静的記憶域を持つプログラムに対して出力されるセグメントは**図 8.37** に示されているようになる．

トランスファ・ベクトルは内部静的領域に置かれる．この領域はプログラムの一部である．これが IBM 360 型計算機で行なわれている方法である．しかし MULTICS では，(1)すべての手続きは純粋（自分自身を変更しない）でなければならず，また，(2)同一のセグメントでも各利用者の番地空間の中では異なったセグメント番号を持つので，純粋な主プログラムの中に内部静的領域の部分を含むことはできない．したがって，MULTICS における外部静的領域に対するポインタは，**リンケージ・セグメント**と呼ばれる別のセグメントの中に格納される．

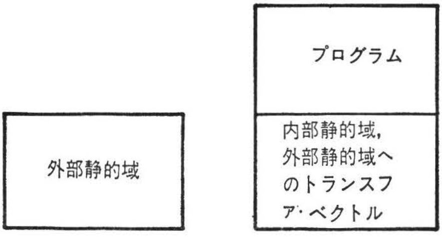


図 8.37 静的記憶域

8.6.2 自動記憶域

自動記憶域は実行時にその手続きが呼出されたとき割付けられる．したがって，コンパイラは各手続きに対して，通常のコードに先行して，手続きの中で

自動的と宣言されている記憶域を割付けるための**導入部**を生成しなければならない．プログラムの最初に現われるコードは，前節の回帰呼出しで使用された SAVE マクロのコードと同様のものである．このようにすると自動記憶域を持つ手続きに入ったとき，割込みの状態，レジスタ，およびすべての自動変数の値を格納するスタック・フレームが作られる（図 8.38）．

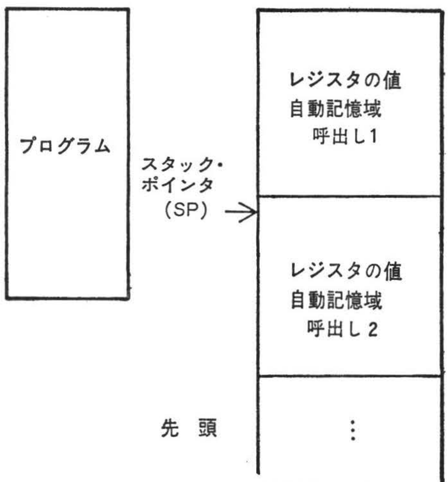


図 8.38　自　動　記　憶　域

スタック・フレームは呼出しの度に割付けられる．これに対してそのスタック・フレームは復帰のときに解放される．1つのスタック・フレームは**スタック・ポインタ**を用いて指定される．スタック・ポインタはフレームの開始番地を指している．自動変数は現在のフレームのスタック・ポインタに相対的に参照される．新しい手続きに入って割付けを行なおうとするときには，単にスタック・ポインタを新しいフレームに設定すればよい．手続きから復帰するときは，単に呼出しを行なったプログラムのスタック・フレームにポインタを戻すだけで記憶域を解放することができる．

8.6.3　被制御記憶域および基底つき記憶域

PL/I の被制御記憶域は自動記憶域と同様に組込むことができる．

ALLOCATE 文は実行時に記憶域を割付けるための導入部として翻訳される（通常，オペレーティング・システムを呼出して行なわれる）．FREE 文は記憶域を解放するコードに翻訳される．

“ごみ集め”（未使用の記憶域を集めること）の仕事を容易にするためには単一のスタックを使用すべきではない．このために塊（heap）と呼ばれる記憶域の構成を使用することができる．たとえば次の文があるとする：

ALLOCATE A;

ALLOCATE B;

FREE A;

被制御記憶域Aに対する領域に，被制御記憶域Bに対する領域が続く．Aが解放されると穴が残る．このとき領域を無駄にすることになる．空き記憶域割付けのパッケージを用意して，これらの領域の一部を移動させて割付けたり，順番には割付けず，穴ができないようなアルゴリズムで割付けたりするような種々の方式が考えられている．

被制御記憶域および基底つき記憶域に対する空き記憶域管理システムはオペレーテング・システムから使用可能の記憶域を受取る．オペレーティング・システムは，手続き ALLOC(長さ) から領域を要求されると，空いている記憶域のブロックを分け与え，ALLOC は指定された長さを持つ未使用ブロックへポインタを設定して復帰する．空いた領域を返却するためには手続き RET（長さ，ポインタ）が使用される．コンパイラは ALLOCATE 文および FREE 文を，適当な引数を用意して手続き ALLOC および RET を呼出すコードに翻訳する．内部被制御域に対してポインタは内部静的領域に設定される．外部被制御域に対してポインタは，他の手続きからも参照可能なように，外部静的領域に設定される．

8.7 ブロック構造

PL/I や ALGOL などの多くの言語ではブロックとブロック構造の使用が可能である．ブロック構造とは変数の有効範囲を指定する方法であり，利用者は別の目的に同じ記号を用いることができる．ブロックは FORTRAN の DO

文や MAD の THROUGH 文と同様に**入れ子**にすることができる．ブロックは記号定義の有効範囲を指定する．たとえば，**図 8.39** の記号の有効範囲は**図 8.40** に示されるようになる．

ブロック構造に関してコンパイラが処理しなければならない問題は 2 つある：

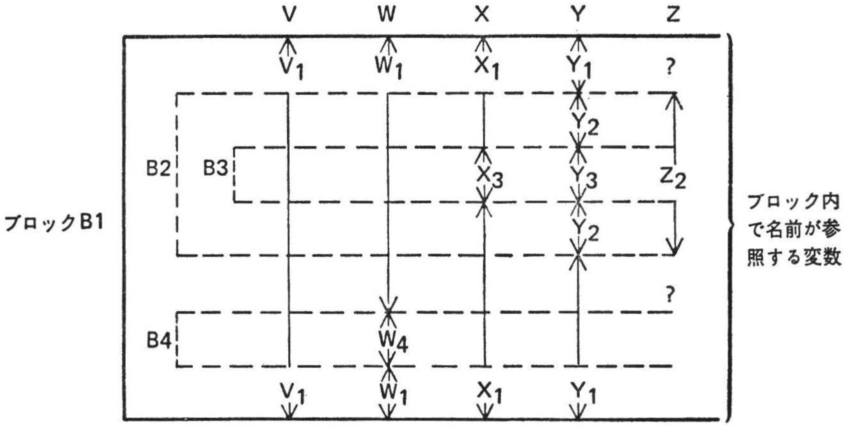
1. 別のブロックの中で宣言されていれば，同じ名前を持っていても，異なった宣言と属性を持つ全く別の変数を指定することができる．コンパイラは，与えられた記号がどちらを指しているのかを決定しなければならない．たとえば，**図 8.39** の文 Y=W でプログラマは B1 のYか，B2 のYか，または B3 のYのいずれを指しているのだろうか．
2. ブロック構造に関する第2の問題は，ブロックの中で定義された自動記憶域をブロックに入ったとき割付け，ブロックから出るとき解放しなければならないことである．ブロック構造は，静的記憶域や被制御記憶域に対しては大きな問題を生じない．

```
B1:   BEGIN;          DECLARE (V,W,X,Y) FIXED BIN(15) AUTOMATIC;
                        =
                      B2: BEGIN; DECLARE(Y,Z) CHAR(8) AUTOMATIC;
                        =
                          B3: BEGIN; DECLARE(Y,X) BIT(16) AUTOMATIC;
                              Y = W;
                              END;
                        =
                          END;
                      B4: BEGIN; DECLARE(W) FIXED DEC(4) AUTOMATIC:
                        =
                          END;
      END;
```

図 8.39 ブロック構造の例

8.7.1 ブロック構造に関する参照情報

まず，1つの記号によって変数が参照されたとき，その変数を探すための問



ただし、V_1、W_1、X_1およびY_1はB1で定義された
V、W、X、およびYに対応する
Y_2およびZ_2はB2で定義された
YおよびZに対応する
X_3およびY_3はB3で定義された
XおよびYに対応する
W_4はB4で定義された
Wに対応する
?は未定義であることを示す

図 8.40 変数の有効範囲

題点と解決法について概説しよう．コンパイラが単語段階で1つの記号を見出したとき，次に続く段階に不偏記号を渡さなければならない．その不偏記号は記号表の中の1つの欄を指す．どの欄を指すのだろうか．同じ名前の記号に対して記号表の中に2つの欄が存在するかも知れない．たとえば，前述のブロック構造において，記号Yはブロック B1, B2 および B3 の中で定義されている．B1 ではYは2進数である．B2 では文字列，そして B3 ではビット列である．単語段階では，宣言やブロックについて何もわからないから，これらが別々のYであると解釈して不偏記号を渡すことができない．すなわち，単語段階でできることといえば記号表に変数の名前を記入することだけである．

1つの組込み方法は，各ブロックの中で宣言された変数に対して記号表の中に新しい欄を作るような解釈段階を用意することである．この段階では定義されているブロックと各記号との対応をつけなければならない．したがって，変

数が参照されたときは，このブロックまたはこれを含む最も内側のブロックの中で定義されている記号を記号表の中で探し，次にこの記号を適当な不偏記号に置換える．この機能を組込む方法を前のモデルを用いて簡単に示そう．**図 8.39** のプログラム構造に対して**図 8.41** に示したように，記号表を記号の名前を記入するものと記号を定義するものとの２つに分けると処理のために都合が良い．

単語段階では名前表に欄を作る．解釈段階でブロックの中に宣言文を見つけると，記号表に欄を作る．各欄には名前表の中の記号の名前を指すポインタを設定しておく．この記号がどのブロックの中で定義されているかを示す情報も与える．また，他のブロックの内で同じ変数を再定義している記号表の欄への指標をも記入する．このようにするとブロックの中で１つの変数が参照されたとき，解釈段階ではそれを不偏記号に置換えることができる．変数が出てくると記号表を探して正しい欄を求め，ブロックを定義した情報と次の定義を示す指標を用いて正しい不偏記号を決定する．

名前表

	名前
1	V
2	W
3	X
4	Y
5	Z

記号表

	名前へのポインタ	属性	次の定義への指標	定義されたブロック	番地
1	V_1		0	1	
2	W_1		9	1	
3	X_1		8	1	
4	Y_1		5	1	
5	Y_2		7	2	
6	Z_2		0	2	
7	Y_3		0	3	
8	X_3		0	3	
9	W_4		0	4	

図 8.41　ブロックに対する記号表の構造

宣言文を用いずに記号が宣言される場合がある．たとえば，文ラベル定数がそうである．

L : /* これはラベルの宣言である */ ;

しかも，この宣言は必らずしもラベルが使用されたり参照される前に行なわれなくてもよい．ラベルは解釈段階で名前表に記入される．構文段階でラベルが出てきたとき，このブロックに対して記号表の欄がなければ，記号表に欄を作らなければならない．現在のブロックに対して1つの欄が存在し未定義であれば，それをラベルであると定義する．現在のブロックに定義された欄が存在すれば，多重にラベルを定義しているので誤りとなる．1つのブロックの中で定義されている変数への参照は，記号表の中の要素を指す不偏記号によって示される．ブロック構造が許されるときは，この記号が宣言されているブロック番号を行列の欄に記入する．ここで示されている設計においては，コード生成段階で記号表の中の欄を求めて参照を処理する．この欄は変数が定義されている最初のブロック（このブロックかも知れないし他のブロックかも知れない）を指す．そして，どのブロックで行列のこの欄が現われているのかを示す．このようにすると，参照されるべき適当な記号を見出すことができる．

8.7.2 ブロック構造に対する記憶域割付け

ブロック構造に関する2番目の問題点は，ブロックの中で定義されている自動記憶域の割付けである．

ブロック B1 に入ったとき，B1 で定義されているすべての変数はそれらに割付けられた記憶域を持っていなければならない．B2 や B3 の中の変数に対しても，これらのブロックに入ったとき同じことがいえる．B3 から，そしてその次に B2 から外へ出るとこれらの記憶域は解放されなければならない．最後に B4 に入ったとき，その記憶域が作られなければならず，B4 と B1 から出るとそれらの記憶域は解放される．この様子は**図 8.42** に示されている．

ブロック構造に対する自動記憶域の組込みは，第8.6.2節で示されたように，サブルーチン呼出しの場合の自動記憶域に対して新しいスタック・フレームを作って割付けるのと同様に行なうことができる．1つのブロックに入る

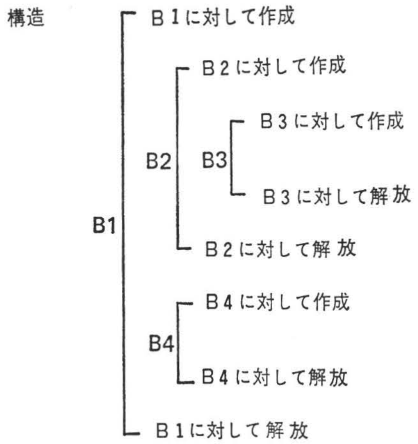


図 8.42 ブロック構造に関する記憶域割付け

とき，外側のブロックに含まれるすべての変数の記憶場所とこのブロックの中の変数に対する記憶場所とを持つスタック・フレームを作ることができる．新しいブロックに入るには，外側のブロックに含まれるすべての変数を格納し(実際は前のスタック・フレームを複写する)，次にこのブロックで再定義されている変数をフレームに含めて新しいスタック・フレームを作る．ブロックから出るときは，単にスタック・フレームを消去する．しかしながら，ブロック構造に関しては，ブロックに入ったり出たりする順番が明確にわかっているので，一般の手続きに対するような柔軟性と一般性を必要としない．手続きの場合には，どの手続きがどれを最初に呼出すかとか，手続きがどのように復帰するかなどはわからない．それに比べてブロック構造はそれほど柔軟でないので一般的に組込む必要はなく，多少は効率の良い記憶域割付けを行なうことができる．言語が配列の次元として定数のみを認めているならば，自動記憶域に対して**図 8.43** に示されるような簡単な割付け方式を用いることができる．

ブロック B1 に入るとき，B1 と B1 より内側のブロックの中のすべての変数に対して最大の記憶域を割付ける．すべての次元が定数であれば実行時にどれだけの大きさになるかを正確に知ることができる．したがって，実行時に正しい容量の記憶域を割付けるコードをこのプログラムの先頭に生成しておくこ

<table>
<tr><th>B1に入ったとき</th><th>B2に入ったとき</th><th>B3に入ったとき</th><th>B4に入ったとき</th></tr>
<tr><td>V_1
W_1
X_1
Y_1</td><td>V_1
W_1
X_1
Y_1</td><td>V_1
W_1
X_1
Y_1</td><td>V_1
W_1
X_1
Y_1</td></tr>
<tr><td rowspan="2">一時的に
使用せず</td><td>Y_2
Z_2</td><td>Y_2
Z_2</td><td>W_4</td></tr>
<tr><td>一時的に
使用せず</td><td>Y_3
X_3</td><td>一時的に
使用せず</td></tr>
</table>

図 8.43 ブロック構造を持つプログラムに対するスタックの割付け

とができる．B2 および B3 に入るときには，B1 に対して割付けられた領域の中で，B2 および B3 の中のすべての変数の場所を確保するようにコードを生成する．ブロック B4 に入るときには，このブロックの中で定義されている変数は B2 および B3 にて定義されている変数が使用していたのと同じ領域に割付けられる．B1 から出るときこの記憶域をすべて解放する．

もし，外側のブロック B1 で計算されたNに対してブロック B2 の中で配列 Z(N) が定義されるならば，B2 に入るときにはZに領域を割付けるために十分な情報を持っている．前の例では，最も外側のブロック B1 に入る前に，コンパイラは B1 および B1 より内側のすべてのブロックに十分なだけの記憶域を割当てていた．必要量の最大値が確保されているとわかっていたので，単に

この記憶域を使用すればよかった．しかしながら，今の場合にはコンパイル時に次元がわからないし，B2 に入るまでは，ブロック B1 の導入部を実行しているときでさえもわからない．

可変の次元を持つ配列を認める言語では自動記憶域に対して，**図 8.44** に示すように，別の方式をとらなければならない．

図 8.44 は，ブロック B1 を含む手続きが呼出されたとき作られたスタック・・フレームを示している．この呼出しによってブロック B1 の中で定義された自動記憶域が割付けられ，それに続いて，B1 に含まれるブロックが持つ記憶域を指すポインタが置かれる．これらのことはすべて，実行時に手続きが呼出されたとき，コンパイラが生成したコードにより実行される．このコードは B1 に対して記憶域を割付け，**表示領域**を設定する．B2 に対して生成されたコードはこの自動記憶域へのポインタおよびこの領域の長さを表示領域に複写する．ブロック B4 に入ったとき，生成されたコードは，B2 および B3 の記憶域の上に記憶域を重ねて配置し，それにしたがって表示領域も変更する．

自動記憶域に対しては実行時にその記憶域に関してある種の情報が必要である．たとえば，配列の大きさはこの配列が割付けられる前に決定されなければ

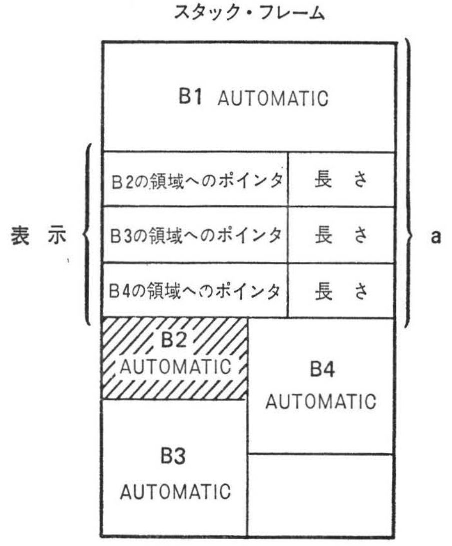


図 8.44 ブロック構造に対する記憶域割付けを指示する表示情報の使用

ならない．もし変数の初期化が必要ならば，この記憶域の中で行なわれなければならない．したがって，自動記憶域の中のすべての変数に対して，初期化される変数とそれらの初期値，配列の大きさなどの情報をどこか別の場所に持っていなければならない．自動記憶域の経費をきわめて大きくしているのは，この情報を格納するためのオーバヘッドである．配列の大きさ，配列の位置へのポインタ，および配列の次元に関する情報は静的記憶域の中に保存される．この情報は**配列情報ベクトル**（array dope vector）と呼ばれている．自動記憶域として定義されている記号列に対応しては**記号情報ベクトル**が作られる．3番目の型の情報ベクトルは PL/I の IBM 360 への組込みで使用され，**記号列配列情報ベクトル**と呼ばれている．実行時に自動記憶域が割付けられると，これらの初期化のための値は新しく作られた記憶域に複写される．したがって，初期化が行なわれるときには，自動記憶域に対して2倍の記憶域が必要となる．

オペレーティング・システムは，記憶域の割付けを強力に援助する機能を持つことが多い．MULTICS システムにおけるセグメンテーションは記憶域の割付けをきわめて容易にしている．

8.8　非局所的な制御の移行

PL/I ではラベル変数を引数としてサブルーチンに渡すことができる．図 8.45は非局所的な GO TO の例である．図には P1 で定義されたラベル変数を使用する手続き P1, P2, および P3 が示されている． P1 はラベルを引数として P2 に渡し，P2 はこれを3番目の手続き P3 に渡す．P3 はそのラベルに制御を移す GO TO を持っている．

PL/I の言語仕様によると，1つの手続きに制御を渡す非局所的 GO TO はラベルが設定されたとき実行していた状態に手続きを復元しなければならないと規定されている．したがって，P1 はラベル変数 LABZ の値がラベル定数 LAB に設定されたときと同じ状態に復元されなければならない．すなわち，そのラベルへの GO TO が実行されるとき，対応したスタック・フレームへ**巻き返す**ことができるように，ラベルに値が割当てられる度に，ラベルとその割当てを行なった手続きに関連したスタック・フレームの位置を記憶しなければならな

```
P1: PROC;
        IF N = 3 THEN CALL P1;
        LABZ = LAB;
        CALL P2(LABZ);
LAB:    X = 1;
        END;
-----------------
P2: PROC (LABX);

        ═══

        CALL P3(LABX);
        END;
-----------------
P3: PROC (LABY);

        ═══

        GO TO LABY;
        END;
```

図 8.45 非局所的制御移行を行なうプログラム

い.

8.9 割 込 み

PL/I の**割込み**は正規の手順には従わない制御の流れを規定する手段である. ある種の**条件**が発生すると, 割込みが通知される. プログラマは, ON OVERFLOW X=Y のように, ON 文を用いて, これが起こったとき取られる動作を規定することができる. 割込みが発生するとモニタ・システムが呼出される. システム割込みルーチンは文 X=Y を実行した後に割込まれたプログラムへ復帰するように組込まれる.

8.10 ポ イ ン タ

PL/I ではポインタとポインタ変数を使用することができる. ポインタは変数を指定する手段である. たとえば, 文 P→A=4; は, P の指定している変数が 4 に等しく設定されることを示している. (基底つき記憶域の場合には多くの

Aが存在する．どのAを用いるのだろうか．それはPが指しているAである．このようなポインタの組込みは，利用者のポインタ情報を単にそのまま用いて参照するようにすればよい．）ポインタで指定される変数を参照するために生成されるコードには，あるベース・レジスタにポインタを設定するコードがそれに先行して置かれる．その記憶域への参照はベース・レジスタの内容を使用する．たとえば，IBM 360ではすべての参照に対して次のコードが生成できる．

```
L   15,"ポインタ" /* 記憶域へのポインタをベース・レジスタに設定する */
L   AC,オフセット(15) /* レジスタACに参照された番地の内容を設定する*/
```

2番目の命令は参照された変数をレジスタACに設定する．オフセットはコンパイル時に定めることができる．また，インデックス指定の場合には実行時に計算することもできる．

すべてのクラスの記憶域に対する参照は，暗にポインタによって指定されていると考えることができる．たとえば，内部静的記憶域はこの記憶域を指しているポインタ IS をつねに持っていると考えることができる．したがってAを内部静的と宣言すると，次のような文を書くことができる：

IS → A＝4；

この文はA＝4を意味する．どのAかというと，内部静的領域の中のものである．

自動記憶域は，スタック・ポインタがポインタであるような条件を持つと考えられる．

SP → A＝4；

この文では変数Aが4に等しく設定される．どのAかというと，スタック・フレームの中のものである．同様に続けると，Aが被制御記憶域と宣言されていると，文 A＝4；は文 CP→A＝4；と考えることができる．

したがって，上のようなコードですべてのクラスの記憶域に対してはローダが“ポインタ”の値を設定し，自動記憶域および被制御記憶域に対してはオペレーティング・システムの記憶域管理ルーチンがポインタの値を設定する．

能率向上のために，1つのクラスの記憶域が頻繁に参照されるならば，実行時にレジスタに設定されたポインタを持つのがよい．たとえば，1つのレジスタはつねにスタック・ポインタとして自動記憶域を指し，他のレジスタには内部記憶域の底を入れるようにすることができる．基底つき記憶域の場合には，作り出される記憶域に対してポインタを設定するよう利用者が指定し，システムが値を設定する．

8.11 まとめ

本章のモデルはきわめて一般的なものである．このように組込まれるコンパイラは，単にコード生成則を変更するだけで出力コードを変更することができる．入力の言語は構文変形則を変換することによって変更することができる．実際のコンパイラでは，上述の柔軟な手法の幾つかは効率上の理由から用いられていない．しかしながら，コンパイラの設計は本章のモデルの段階を追うべきであると考える．組込んだ後で，余分の柔軟性を取り除けばよい．たとえば，構文変形則やコード生成則をコンパイルする．

新しい言語に関する難しい問題点は算術式の解析ではなくて，記憶域の割付けや参照方法である．

コンパイラ自身および出力コードはオペレーティング・システムと関連していなければならない．特に，条件処理や記憶域のクラスはオペレーティング・システムと深く関係する．コンパイラはオペレーティング・システムをよく知り，システムの管理ルーチンを上手に利用するための呼出し手順のコードを生成しなければならない．

演習問題[1]

第1部

1. コンパイラを構成する段階のブロック図を描き，各段階の主要な機能を述べよ．
2. インタプリタとトランスレータの基本的な相違点は何か．コンパイラはトランスレータであると考えられる．コンパイラの段階の内でインタプリタと考えられるものがあるか．それはどれか．
3. 次の相違点は何か．例をあげて説明せよ．
   段階とパス
   構文解析と意味の解釈
   トークンと不偏記号
4. 標準システムのアセンブラを用いずに，コンパイラの中でコンパイルされたプログラムのアセンブルを行なうことの利点と欠点とを述べよ．
5. 不偏記号を用いることの目的は何か．不偏記号はなくすることができるか．できるとすればその得失はなにか．
6. 段階は，順番に処理され，はっきり区別できない部分である．それらは互に関連する．たとえば単語，構文および解釈の段階は密接に関連している．この3つを1つの“行列生成”段階で置換えてしまわない理由は何か．
7. どの段階が機械に独立なデータ・ベースを使用するか．
8. プログラムを最適化することはつねに意味のあることか．

第2部

9. 生成則の使用と変形則の使用との相違点について詳細に説明せよ．
10. コンパイラはモジュール化された“差し込み”ユニットのつながりである

---

†は計算機固有の問題であることを示す．

ことを強調して設計を論じてきた．このような設計法を行なうと，言語あるいは機械に関係する機能とそれらに独立の機能とを分離することができる．次に示したデータ・ベースに対して，適当な属性を✓で示せ．**註**：データ・ベースは，構造と内容が共に機械または言語に独立であると考えている．

	作成される表	固定の表	言語に関係する	機械に関係する	言語に独立	機械に独立
終端記号表						
記号表						
リテラル表						
変形則						
不偏記号表						
行列						
行列の演算子						
コードの生成則						
活動ルーチン						

11. IBM 360 で実行する PL/I コンパイラがあり，HIS 645 で実行するためのコードを生成しようとする．このコンパイラが本章のモデルに従って作られているとすると，いかなる変更が必要か．コンパイラは IBM 360 で実行されるものとする．

12. どの段階で行列に追加を行なったり修飾を行なったりするか．また各々は何をするか．どの段階で行列を修飾せずに参照するか．

13. コード生成段階で次の文に対してコードを生成するためには，記号表にどのような情報が記入されていなければならないか．

x＝A．B．C(50)

14. 次の変形則で誤っているのはどこか（変形則3は表の下の方で出てくると仮定している）．

```
    /// * /
1： 〈記号〉 / ″ / * / 2
    〈演算子〉 // S1 / 1
2： / procbgn / * / 3
      ⋮
```

15. “固定”の表と“作成”される表の例を示せ．

16. 言語を記述するには多くの方法がある．正準体系による記述と，変形則に

よる記述と，BNF による記述との基本的な相違点は何か．

17. DCL 文と PROCEDURE 文に対する行列の欄を示せ．

18. 本書では“＋”演算に対するコード生成の原形（マクロ）の例を幾つか示している．これらの原形はだんだん複雑となり，最適化の技法も高度になっている．“－”演算に対するコード生成の原形を示せ．答えに対する評点は最適化の技法の程度に関係するが，複雑な方法の一部を示すより１つの方法を詳しく説明する方がよい．IBM 360 に対するコードを生成するものと仮定せよ．

a. どの情報を参照できると仮定しているかを一覧表で示せ．たとえば，レジスタの内容．

b. 減算演算子に対するコード生成アルゴリズムの流れ図を示せ．

c. “－”マクロを IBM 360 マクロ言語で定義せよ．

d. コード生成が完了したときどこに出力が残されるのか述べよ．

19. 機械に関係する最適化の例を３つあげよ．これらの型の最適化がコンパイラのどこで行なわれるかを述べよ．また，最適化をそれ以前またはそれ以後の段階で行なうことの得失を説明せよ．

20. 共通項の削除はどの段階で行なわれるか．その理由は何か．

21. 最適化の形式には多数あり，各形式は最終的なコードの生成に少しずつ寄与する．ある形式は他よりも一般的であることもある．型でそれらを区別できれば都合が良い．**次の表**の各列には最適化の型が記入してあり，各行には PL/I の文が対応している．

a. 表の各列に示した文に対して適用可能の最適化に✓を記せ．２つ以上の最適化も可能である．

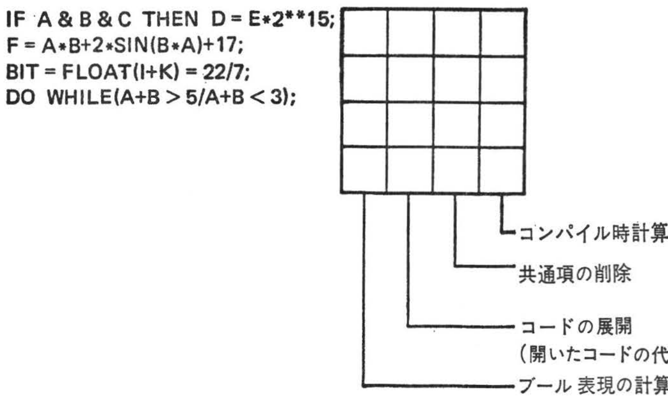
b. 上の最適化の各型はどの段階で実行されるか．その理由は何か．

22. **共通項の削除**―― －(A－B) と B－A が等しいということを確かめるのはきわめて容易である．しかし，第 8.2.4.2 節のアルゴリズムにはこれが含まれていない．上述の場合を含むように修飾せよ．

23. A, B, C が宣言 FIX BIN を持っているとする．文

A＝B＋C;

は行列の中に次のように記され；



i) + B C

i+1) = A T_i

次のようなコードを生成する：

```
 L     0, B
 A     0, C
[ST    0, Ti]
[L     0, Ti]
 ST    0, A
```

（角かっこに囲まれた命令はコードの最適化によって削除できる）

しかし，多くのコンパイラは同じ論理演算で幾種かの異なったデータの型を用いることができる．たとえば，次の宣言を考える：

```
DCL    (A1,B1,C1)  FIXED BINARY;
DCL    (A2,B2,C2)  FLOAT BINARY;
DCL    (A3,B3,C3)  FIXED DECIMAL;
```

次に示した一見同じように見える文は全く異なったコードを生成する（最適化していない）：

A1=B1+C1;		A2=B2+C2;		A3=B3+C3;	
L	0,B1	LE	0,B2	MVC	TEMP,B3
A	0,C1	AE	0,C2	AP	TEMP,C3
ST	0,T_i	STE	0,T_i	MVC	T_i,TEMP
L	0,T_i	LE	0,T_i	MVC	TEMP,T_i
ST	0,A1	STE	0,A2	MVC	A3,TEMP

a. 上に示した新しい型の算術演算を処理するためには MINI-PL/I（図 8.17）の変形則にどのような修飾を行なえばよいか．不必要とすればなぜか．

b. 活動ルーチン（図 8.18）にはどのような修飾を行なえばよいか．不必要とすればなぜか．

c. 行列の中で使用される演算を修飾する必要があるか．その理由は何か．

d. コード生成段階とマクロ定義を修飾する必要はあるか．この段階のプログラムに必要となる変更を述べよ．

e. “混合モード”の式，すなわち，演算数が同じデータの型でない式がコンパイラで許されると仮定する．上に行なった修飾のほかに，この機能を組込むために必要となる修飾を述べよ．

24. 図 8.17 および図 8.18 の変形則と活動ルーチンを使用し，コンパイラの単語段階，構文段階，解釈段階およびコード生成段階を模擬して，次の各々のプログラムをコンパイルせよ．

1)
```
BEGIN: PROCEDURE (A,B);
DECLARE (A,B) FIXED BINARY (15) STATIC;
A = 10;
B = A;
END;
```

2)
```
MIT: PROCEDURE (JOHN, STU, BILL);
DECLARE    (JOHN, STU) FIXED BINARY (31)
           STATIC, BILL FIXED BINARY (15)
           STATIC;
JOHN = STU * BILL;
STU = JOHN - BILL * 10;
RETURN (STU);
END;
```

a. 単語段階，解釈段階，および記憶域割当て段階の後の記号表およびリテラル表の内容を示せ．

b. END がスタックに入れられた直後のスタックの内容を記せ.

c. 変形則をそれらが解釈される順番に記せ.

d. 解釈段階がすんだ後での行列の内容を記せ.

e. 行列のマクロによって生成されるアセンブラ語のコードを記せ.

f. 上の変形則と活動ルーチンを修飾して, ++ または +* が現われたとき, "error" を呼出すための誤り検出機構を組み入れよ.

25. 次の FORTRAN 文を考える:

DO17I=1, 10, 1　　(FORTRAN の DO 文)

FORTRAN プログラマならわかるように, 等号の左には3つのトークン(記号) がある. それらは DO, 17, および I である. これらは完全に区別可能である. しかし, 次の正しい FORTRAN の算術文がある:

DO17I=1

したがって, "," が出てくるまで (または "," がないことがわかるまで) これら2つの文は区別ができない. 後の場合には DO17I は1つのトークンである.

a. コンパイラのどの段階にこのあいまいさが影響を与えるか. その理由は何か.

b. 上の2つの文に示された問題を解決するにはどうすればよいか. 影響のある段階はどのように変更されなければならないか示せ.

26. 行列の生成とマクロ展開を行なわずに, 活動ルーチンが実際の機械語の2進コードを直ちに出力するような高速の '1パス' コンパイラを書くことが可能である. 本章のコンパイラ設計から見て, この方式の主な欠点は何か.

27. a. 演算子+, -を持つ算術式を解析するアルゴリズムの流れ図を描け. 演算子の優先順位を用いよ (問7.1参照).

b. 変形則だけを用いて上の (a) を組込め.

c. 単一の PL/I ルーチンで上の (a) を組込め.

28. †変形則に対する解釈ルーチンを組込め.

29. MIN-PL/I に対する活動ルーチン (図 8.18) を組込め.

30. †コード生成ルーチンを書かずに, 行列によって処理を行なう解釈ルーチンを組込め. 行列の演算子は図 8.18 に示した MINI-PL/I の活動ルーチン

とする.

31. †次を用いて MINI-PL/I を組込め. PL/I プログラムの例を実際にコンパイルしてみよ.

a. 第6章問11の単語解析プログラム

b. 本章問28の解釈ルーチン

c. 本章問29の活動ルーチン

d. 本章問30の行列の解釈ルーチン

第 3 部

32. 自動記憶域または被制御記憶域として宣言された変数に関して，どのような情報をコンパイルの中の実行プログラムに渡さなければならないか. この情報は記号表の中にある. この情報を渡さねばならない理由は何か.

33. 回帰性の問題を解決するには自動記憶域を使用すれば十分であるか. (手続きに入る度にすべての自動変数に対して新しく記憶域が割付けられ, RETURN 文や END 文がくると最も新しい記憶域が解放される.) その理由は何か.

34. 本書では誤りの検査やその処理について論じなかった. それは主題を散漫にするからである. 次に示す各誤りの検査と処理のために，コンパイラのどの部分またはどの段階を修飾する必要があるか. なぜその部分を選んだのかを述べよ. そしてさらに誤り検査を続けるためにはどのような修正を行なえばよいかを述べよ.

a. すべての定数の添字が正しい添字の範囲にあることの検査.

b. 記号の正当性の検査. たとえば，長さ，正しさなど.

c. PL/I や ALGOL の IF 文には見出し語 “THEN” が必要である. IF 文に見出し語 THEN がないときこれをどのように処理するか.

35. この問では図 8.16 に示した PL/I のサブセットを修飾して次の形式の DO 文を追加する.

1) DO {変数}={表現 1} TO {表現 2} BY {表現 3};

2) DO;

a. まず，新しい形式を含むように**図 8.16** の BNF による定義を修飾せよ.

b. DO 文を扱うために**図 8.17** の変形則を修飾せよ.

c. 次に固定のデータ・ベース（**図 8.13**）を考える. 新しい文を組込むために修飾せねばならないのはどれか. どの手続きを変更しなければならないか.

d. 単語段階，構文段階および解釈段階に対して必要な変更を述べよ. 単語段階で変更が必要ならその変更の仕方を，新しい活動ルーチンが必要ならばそれを，また，すべてのデータ・ベースに対する影響についても述べよ.（注意：DOグループは許される.）

e. 次のように定義される"ADD 文"について問 a より d に答えよ.

ADD {変数１}, {変数２}, …, {変数 n} TO {変数 p}；

TO の前には（１個以上）任意個の変数があってもよい. これらすべては TO の後に示される１つの変数に加算される.

36. どの型の宣言文がアセンブリ・コードに現われるか.

37. 今考えている PL/I では次のような基底ポインタを限定詞として使用できないので拡張を考える.

(P1→A.B.NEXT)→A.B.C

a. この機能の組込みにどのような問題が起こるか.

b. この問題を解決するにはどのような修飾が必要か.

38. プログラム言語においてある種の高度の機能はコンパイラの領域を越えるものがある. 次の項目に対してコンパイラとオペレーティング・システムの境界（インタフェース）を論じよ.

a. 被制御／基底つき記憶域

b. 条件とシグナル

c. I/O

39. †完全な PL/I コンパイラを組込め.

第 9 章

オペレーティング・システム

第1章で述べたように，利用者のオペレーティング・システムに対する見方は様々である．オペレーティング・システムの果たすユーティリティの機能を重視する利用者もあれば，主として使用されている制御言語と各種の設備という面を重視する利用者もある．本書が目指しているシステム・プログラムの観点からは，オペレーティング・システムの構造や効率が大事である．これらの要素は，これから注意を向けようとする**資源管理**に関係する問題である．

本章では，第2.1節で示した一般的な計算機のシステムの構造をより詳細に説明し，その能力をより詳細に調べる．これには I/O プログラミングの議論も含まれる．オペレーティング・システムによって制御される4つの主要な要素は**図9.1**に示したように，

(1)**記憶装置**，(2)**プロセッサ**，(3)**入出力装置**，および (4)**情報**である．

資源管理という見方によって，各種のオペレーティング・システムの設計や組込みを組織的に研究することができる．

資源管理，したがって，オペレーティング・システムの統一的な理論ができかかっている．同じ理論や技法が記憶装置，プロセッサ，入出力装置，および

	資　源	例	管理を行なうソフトウェアの例
1	記憶装置	コ　ア	記憶管理，ページング
2	プロセッサ	CPU, I/Oチャネル	トラフィック制御，スケジューラ
3	入出力装置	テープ，ドラム，カード穿孔機	スプーリング
4	情　報	セグメント（システム，利用者，ライブラリ・セグメント）	ファイル・システム，ライブラリ，セグメント管理

図9.1 システムの資源

情報の管理の問題に適用できる．記憶装置のスケジューリングとプロセッサのスケジューリングとでは同種の問題が生じる．ごく最近になって，記憶装置のスケジューリングに都合のよいハードウェアが使用可能となり，記憶装置を分割して，利用者にその一部だけを割付けることができるようになった．プロセッサに対しては，同様なハードウェアの柔軟性はまだ得られていない．たとえば，算術演算部を１人の利用者に，論理制御部を他の利用者に割当てるということは今のところできない．

第 1 部

9.1 I/O プログラミング：多重プロセッサと割込み機構

前章までは，主として各種のプログラミングのための道具，特に，アセンブラ，ローダ，およびコンパイラのためのアルゴリズムを発展させてきた．簡単のために，プログラムの命令を順番に実行して行く１台の CPU を持つ計算機システムを仮定してきた．そして計算機システムの多くの“目立たない”しかも複雑な面，たとえば，I/O プログラミング，I/O プロセッサ，多重プロセッサ，並列計算，および非同期割込みなどを無視してきた．さらに，システム効率の最適化，**スループット**（一定の時間内に達成される“仕事”の量）に関する問題点を，コンパイラの最適化を簡単に論じた以外，ほとんど何も述べなかった．

幸いなことに，スループットの最適化や非同期動作および並列動作に関するほとんどすべての問題は，通常，オペレーティング・システムに局限することができるので，これらを省いても前章までの議論にはあまり影響がない．以下におけるオペレーティング・システムの資源割当ての議論をより完全に理解し評価できるように，まず，前に述べた“単純な”計算機の概念を発展させておく必要がある．

9.1.1 多重プロセッサ・システムの発展

計算機システムには３つの基本要素がある：

1. 中央処理装置（CPU）

2.　主記憶装置

3.　I/O 装置（周辺機器）

初期のシステムでは，これらの要素は**図 9.2** に示したように接続されていた．

中央処理装置と記憶装置は，基本的に第2章に示した一般的な計算機の構造に対応するものである．ADD および SUBTRACT などの演算命令のほかに，I/O 装置を1台ずつ動作させるために，READ A CARD, PUNCH A CARD および PRINT A LINE などの命令があった．初期の計算機では ADD 命令は1ミリ秒かかるのに，READ CARD 命令は 500 ミリ秒（最高1分間にカード約 100 枚の速度）かかった．

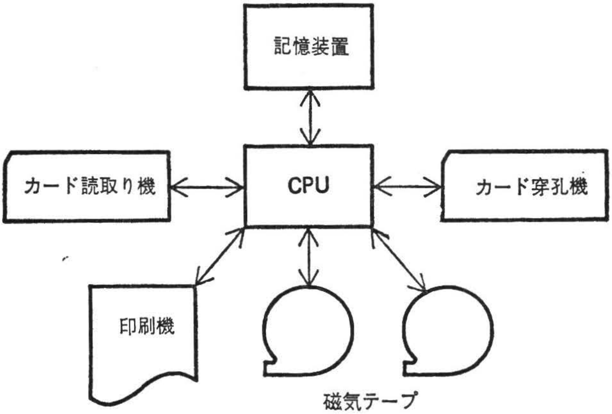


図 9.2　初期の計算機システム

計算機システムが発達するにしたがい，速度と容量が大きくなり実行できる仕事量が増大した．たとえば，容量 1000 語で参照時間1ミリ秒の記憶装置は，容量 100 万語で参照時間1マイクロ秒以下のものに代り，速度も容量も 1000 倍になった．同様に CPU も進歩し，通常の命令実行時間は1マイクロ秒以下となっている．

しかしながら，このような進歩はすべてが歩調を合わせてというようには行っていない．たとえば“高速”カード読取り機や印刷機の速度は1分間にカード 1000 枚または 1000 行（1枚または1行につき 50 ミリ秒）におさえられている．したがって，CPU は，1つの READ CARD 命令実行の間に約 50,000 もの演算命令を実行できる．磁気テープ，ディスクおよびドラムなどの多くの I/O

装置は，カード読取り機や穿孔機，印刷機などに比べるとはるかに速いが，それでも CPU の速度に比べると低速である．

このような I/O 装置と CPU-記憶装置との間の速度の不調和が **I/O チャネル**，すなわち，**I/O プロセッサ**の発達の動機となった．I/O チャネルは I/O 装置を動作させるために作られた特別のプロセッサである．I/O チャネルは単純で，用途が限定され，しかも余り高速でなくてもよいので，普通の CPU よりはるかに低価格のものでよい．**図 9.3** は幾つかの I/O チャネルを持つ計算機システムを示している．すべての入力および出力はチャネルを介して実行されるので，“カードを読込む”ようなことのために時間を費さずに高速の計算を実行できる．さらに，複数台のチャネルを同時に動作させることが可能であるので，多数のカード読取り機，穿孔機および印刷機は同時に動作できる．

I/O チャネルには，基本的な考え方は同じであるが，きわめて単純なプロセッサから完全な中央処理装置に近いものまで種々のものがある．I/O チャネルによっては単一の装置のみを接続する（すなわち，I/O 装置毎に1つのチャネルを必要とする）ものもあれば，幾つかの装置を接続して一時に1つの装置を動作させる（**セレクタ・チャネル**）ものや同時に複数の装置を動作させる（**マ**

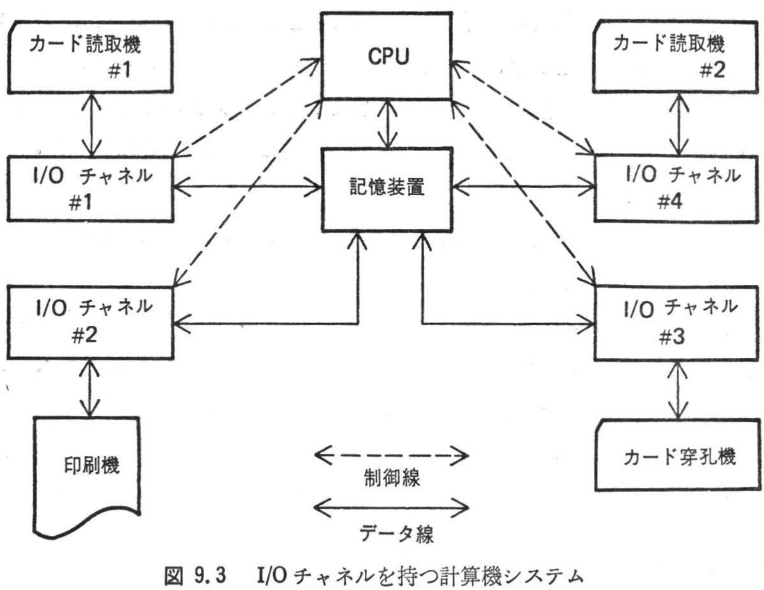


図 9.3 I/O チャネルを持つ計算機システム

ルチプレクサ・チャネル）ものまである．

図9.3に示した計算機システムは，相当な準備をしておかなければ効率よく働かないというものではない．指導者としてすべてのプロセッサを協同作業させるのがオペレーティング・システムの機能である．オペレーティング・システムが取り扱わねばならない問題をよりよく理解できるように，次節においてI/O プログラミングの例と I/O 割込み機構について述べる．

9.1.2 I/O プログラミング

中央処理装置は通常 START I/O および HALT I/O などの特別な命令を用いて I/O プロセッサと通信する．I/O プロセッサは状態レジスタを調べることにより中央処理装置の状態を知ることができる．I/O チャネルと CPU との通信は，通常，割込みを用いて行なわれる．**割込み**とは CPU の実行を中断し，状態を保存し，特定の番地に制御を移すハードウェアの機能である．制御移行先の番地は割込みに応答してその処理を行なうプログラムの番地を示す．割込みの発生の結果実行されるプログラムを**割込み処理プログラム**という．

I/O プロセッサはそれ自身の命令の組を解釈する．これらの命令を組合わせて書かれているプログラムは **I/O プログラム** と呼ばれ，それらを作ることを **I/O プログラミング**という．プログラマは通常自分の I/O プログラムを書かずに，システムに用意された関数を呼出してI/O プログラムを準備させる．モニタ・システムはI/O プログラムを利用者の特定の要求に応じて仕立て，それらを実行させる．

I/O プロセッサを理解するのに，CPU に対して第2.1.1節でとった方法を用いよう．新しい計算機やプロセッサに出会ったとき発する5つの基本的な質問に答え，幾つかの I/O プログラムの例を書いてみる． I/O プロセッサについては，中央処理装置のように命令の標準的な構成法が達成されていない．しかし，その機能は容易に一般化できる．ここでは IBM 370 型システムの I/O プログラムの例を用いる．割込み処理の節では多重処理システムにおける割込み処理を，例として，CPU と I/O チャネルの間の通信に関して述べる．

9.1.3 I/O プロセッサの構造

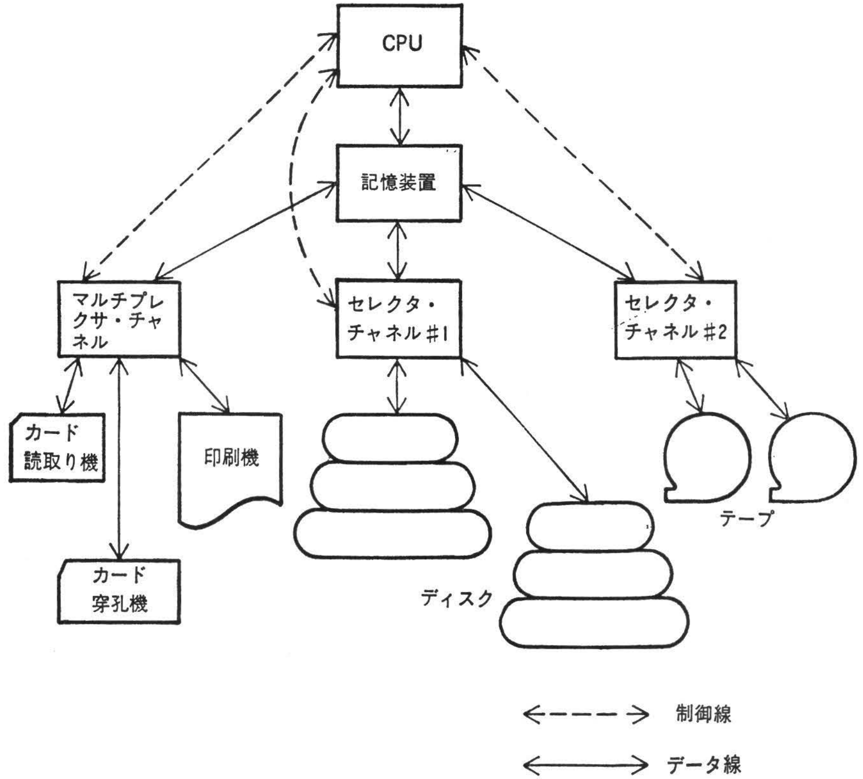


図 9.4 IBM 370 型システムの構成

図 9.4 は IBM 370 型計算機にみられるシステム構成を示す．

IBM 360／370 型 I/O プロセッサに関する 5 つの基本的な問と答えは次に示したとおりである：

A. 記憶装置——記憶装置の基本単位，容量，および番地指定法は何か．

記憶装置は IBM 360 の中央演算装置が使用するものと同一である．基本記憶単位はバイト，容量は最大 2^{24} バイト，番地指定法として I/O チャネルは 24 ビットの絶対番地を使用する．

B. レジスタ——何個レジスタがあるか．容量は，機能は，また相互の関係は何か．

I/O チャネルにはプログラムで利用できるレジスタがない．しかし，プログラム・カウンタとデータ・カウンタがある．I/O 装置によっては割算などの命

令で使用される CPU 内部の作業用レジスタに類似した内部レジスタを持っている.

C. データ――プロセッサによってどの型のデータが処理できるか.

基本的には，たとえば一連の，1から 2^{24} バイトまでの長さの論理データが取り扱える．I/O 装置によってはデータのコード変換を行なうこともある（たとえば，EBCDIC から BCD へのコード変換)．また，I/O プロセッサは I/O 装置の状態を示すセンス・データを使用し，このデータを読んだり処理したりできる.

D. 命令――I/O プロセッサはどのクラスの命令を解釈するか.

I/O 命令（しばしば**コマンド**と呼ばれる）は基本的には次の3種に分けられる：

1. データ転送：読込み，逆方向読込み，書出し，センス（装置の状態を検査する).
2. 装置の制御：制御（改ページ，テープ巻戻しなど).
3. 分岐：チャネル・プログラムの中での制御の移行.

チャネルはチャネル命令（**チャネル・コマンド語：CCW**）を記憶装置から取り出して，次の書式にしたがって解釈する：

命令コード	データ番地	フラグ	未使用	回数
0　7	8　31	32　36	37　47	48　63

命令コードは2～4ビット（ビット数）の命令ビットと4～6ビットの修飾ビットから成っている．命令ビットは標準のものが定まっているが，修飾ビットは各装置によってそれぞれ異なる.

データ番地（ビット8～31(ビット位置)）は主記憶装置の中のバイトの位置を指定する．これは命令によって参照されるデータ部分の開始点である.

フラグは命令をより詳細に規定する．次の5つのフラグがある：

1. **チェイン・データ**・フラグ（ビット32）は，最初のデータ領域の処理がすむと，次の CCW によって指定された記憶域が現在の処理のデータとして使用されることを指定する.
2. **チェイン・コマンド**・フラグ（ビット33）は，現在の動作が正常に完了

すると，次の CCW を引き続いて実行することを指定する．

3. **長さ検査の抑制指定**フラグ（ビット34）は，正しくない長さであるとの表示を行なわないことを指定する．

4. **無視**フラグ（ビット35）は，情報の転送を記憶装置に対して行なわないことを指定する．

5. **プログラム制御割込み**フラグ（ビット36）は，この CCW がチャネルで実行されると，チャネルに割込み条件を発生させる．

回数部は，この CCW で参照されるべき記憶域のバイト数を指定する．

一続きの CCW のかたまりを**チャネル・プログラム**という．CPU はチャネルの命令実行を開始させる．

E. 特殊機能——チャネルは各 I/O 動作に対して，与えられた見出し語を記憶装置のブロックの見出し語と比較して， IBM 370 の正規の情報保護動作をも実行する．

また，チャネルは IBM 370 割込み機構の一部を構成する．これらの点については，次節で CPU とチャネルの間の通信に関して明らかにする．

9.1.4 I/O プログラムの例

例 1:

“読み飛ばし” を行なう I/O プログラムを書いてみよう．このプログラムはカード読取り機からすべてのカードを読込み，最後のカード以外のすべての情報を無視するものである．最後のカードの情報は欄 1 から欄 72 までと欄 73 より欄 80 までの 2 つのデータ部分に分けられ保存される．この I/O 動作は CPU（モニタ・モードで実行している）が現在実行しているジョブに誤りを見つけたときに開始される．オペレーティング・システムは，カード読取り機にある次のジョブに移るために残りのカードを単に読込み，診断のために最後のカードの情報だけを残しておきたい．**図 9.5** はこれを行なうための 16 進数で書かれた命令コードを示している．これらは実際に記憶装置の中に現われる命令である．

図 9.5 の I/O プログラムは 600 番地から始まると仮定している．最初のチャネル・コマンド語は 02 の命令コードを持つ．これは READ と解釈される．こ

番地	命令コード	データ番地	フラグ	未使用	回数
000600	02 ← 読込み	000700	80 ← データ・チェイン	00	0048 (=72_{10})
000608	XX	000800	40 ← 続行	00	0008
000610	08 ← 制御移行	000600	XX	XX	XXXX
・ ・ ・					
000700		カード・データ 72 バイト			
・ ・ ・					
000800		カード番号 8 バイト			

図 9.5　I/O プログラム——読み飛ばし

の命令はカード読取り機から 72 バイトのデータを番地 700〜771 へ読込む．80 というフラグ部は，データの残りの部分は次のチャネル・コマンド語で処理されることを示している．次のチャネル・コマンド語は，カードの最後の 8 バイトを番地 800〜807 に読込むことを指定する．フラグ 40 は引続いて次のチャネル・コマンド語が実行されることを指定している．

最後のチャネル・コマンド語は 610 番地に置かれている． 命令コード 08 は I/O プロセッサによって，データ番地部に示されている 600 番地の CCW に制御を移すものとして解釈される．

このプログラムを実行している I/O プロセッサは各カードの情報を読込み，いつも 700 番地と 800 番地に重ねて記入する．最後のカードを読込んだ後で，I/O プロセッサが I/O プログラムの最初の命令を実行しようとして，カード読取り機にもうカードがないことがわかると，実行を終結してそのことを CPU に通知する．

I/O プログラムのループは，磁気ディスクの情報を探索したりする場合などに多く用いられる．しかし，たいていの単純な I/O プログラムはループを持たない．

例 2:

印刷機でメッセージを印刷し，改ページを行なってから次のページの先頭に見出しを印刷するI/Oプログラムを書いてみよう．このようなI/Oプログラムは利用者のジョブ終了時に呼出される．このI/Oプログラムを用いて印刷したいメッセージは，“END OF PAGE”と次のページの先頭の“TOP OF NEW PAGE”である．**図 9.6** に示したプログラムがこれを行なう．

最初の I/O 命令は 300 番地から始まる場所に格納されている．命令コードは 01 で，I/O プロセッサに 1 行印刷させる．印刷される行は，番地部に示されているように 400 番地に格納されており，11 文字から成っている．

308 番地にある次の命令は，命令コード 8B を持ち，印刷機で改ページを行なう．最後の命令は，命令コード 11 を持ち，500 番地に置かれている長さ 15 の文字列を印刷する．さらに，この命令コードは見出し行を印刷した後に行を 2 行送ることを指定している．

番地	命令					註釈
	命令コード	データ番地	フラグ	未使用	回数	
000 300	01	000 400	40	00	000B	1 行印刷
000 308	8B	XXX XXX	40	00	XXXX	ページを送る
000 310	11	000 500	00	00	000F	見出しの印刷
⋮						
000 400	"END OF PAGE"					データ
⋮						
000 500	"TOP OF NEW PAGE"					

図 9.6 I/O 印刷プログラム

9.1.5 CPU とチャネルとの間の通信

前節まではチャネル自身がどのようにして働くかを見てきたが，本節ではチャネルがどのようにして CPU と通信するのかを調べてみよう．チャネルを設置する理由は CPU をしばしば非常に時間のかかる I/O 動作の制御から解放することにあった．CPU とチャネルは通常主人と奴隷の関係にある．すなわち，CPU はチャネルに‘動作を開始せよ’とか，‘停止せよ’とか，‘実行している

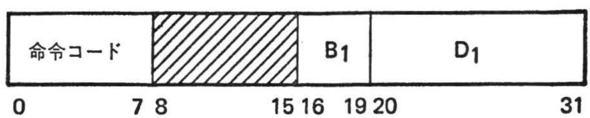
ことを変更せよ’とか命令する．これに対して，チャネルは通常 CPU によって指示があるまでは動作を開始できない．

CPU とチャネルとの間の通信には次の2種類がある：

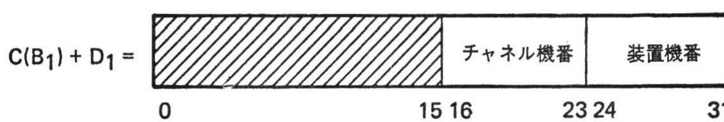
1. CPU によって開始される I/O 命令 (CPU からチャネルへ).
2. チャネルによって開始される CPU への割込み (チャネルから CPU へ).

I/O 開始命令は CPU によって実行される．本節では CPUの I/O 命令とチャネルの I/O 命令との関係を述べる． I/O 割込みの説明は次節にて行なう．

すべての CPU の I/O 命令は次に示す‘修飾された’SI 形式である：



チャネル番号および装置番号はレジスタ B1 の内容と命令の D1 部の内容との和で指定される．和のビット 16〜23 はチャネル機番を示し，ビット 24〜31 はチャネルに接続されている装置機番を示す：



CPU の I/O 命令には次の4つがある（**註**：これらは CPU によって実行される）：

1. START I/O——I/O を開始させるには次の2つの情報が必要である：
   a. チャネル番号および装置番号.
   b. チャネル・プログラムの開始番地.

   START I/O 命令はチャネル番号と装置番号を指定する（たとえば，SIO 0 0E：ここで0はチャネル番号0を，0E は装置番号 0E を示す)．記憶装置の $72〜75_{16}$ 番地は**チャネル番地語** (CAW) であり，チャネル・プログラムの始まりを示す．これについては後に述べる．
2. TEST I/O ——指定されたチャネルおよび装置の状態（動作中か否か）が条件コードに設定されて通知される．条件コードは標準の条件分岐命令によって調べることができる．
3. HALT I/O——指定した I/O 装置とチャネルで現在行なわれている I/O

動作を終結する．

4. TEST CHANNEL ——指定されたチャネルの状態（動作中か否か）が条件コードに設定されて通知される．

記憶装置の小さい方の番地（最小限 0〜128）は特別の目的に確保されている．64〜71 番地は 2 語長のチャネル状態語（CSW），72〜75 番地は 1 語長のチャネル番地語（CAW）である．

チャネル状態語は I/O 装置の状態や，I/O 動作が終了した状態の詳細を示す．CSW は I/O 動作の過程や，START I/O, HALT I/O，および TEST I/O の実行中に設定される．CSW の書式は次のとおりである：

記憶保護キー	0000	コマンド番地	状態	回数
0　3	4　7	8　31	32　47	48　63

9.1.6 割込み機構と処理

割込みとは，命令実行に非同期の信号に応答して制御の移行を起こすことをいう．通常これらの信号を受信すると，プロセッサ（通常 CPU）は，プログラムの状態情報を保存した上で命令カウンタの内容をあらかじめ定められた番地の内容で置換える（または，特定の値を設定する）．割込みに引続いて，プロセッサは**割込みルーチン**の実行を開始する．割込みルーチンは割込みの条件に応答するように書かれたプログラムである．**図 9.7** は割込まれたプロセッサの軌跡の例を示している．この例では割込みが発生したとき，プロセッサはプログラムの点 1 を実行していた．プロセッサは強制的に割込みルーチン 2 に制御を移され，割込みルーチンが終了するとプロセッサは元のプログラムの点 3 から実行を再開したことがわかる．

CPU の“状態”，すなわちプログラムの現在の状態は**プログラム状態語**（PSW）と呼ばれる 2 語長のレジスタに格納されている．これは I/O プロセッサの CSW に相当するものである．PSW は命令実行順序を制御し，現在実行されているプログラムに関してシステムの状態を格納したり指示したりするために使用される．アクティブな，すなわち，制御中の PSW は“現用”の PSW である．割込みの起こったときに PSW を記憶装置に格納しておくことによって，後で CPU の状態を検査したり再ローディングしたりすることがで

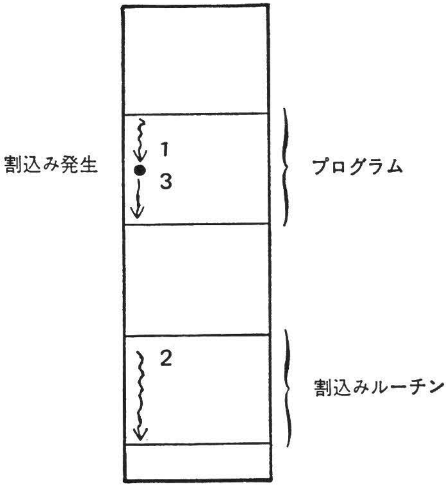
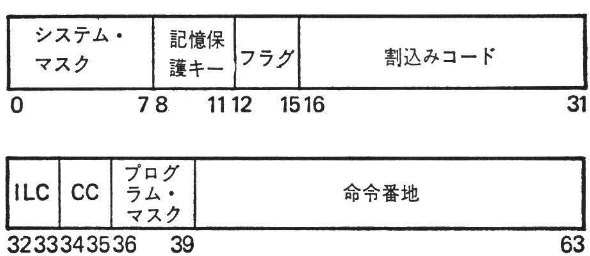


図 9.7　割込みの発生とプロセスの軌跡

きる．新しい PSW，または，それの一部をロードすることによって，CPU の状態を変化させることもできる．PSW の書式は次のとおりである：



（PSW に関する詳細な説明は，IBM マニュアル‘動作原理’を参照せよ．）

PSW の中のビットはある種の割込みをマスクするのに使用することができる．**マスク**されると I/O 割込み，外部割込みおよび機械検査の割込みは一時的に禁止され，保留される．プログラム・マスクは 15 種のプログラム割込みの内の 4 種を無視する．他の 11 種のプログラム割込みおよびスーパバイザ呼出しの割込みはマスクすることができない．

プログラマは 11 種の状態切換え命令を使用することができる．これらには，

PSW のロード（LPSW），プログラム・マスクの設定（SPM），システム・マスクの設定（SSM），スーパバイザ呼出し（SVC），記憶装置見出し語設定（SSK），および記憶装置見出し語挿入（ISK）などがある．

PSW の変更，修飾およびロードを行なう命令は特権命令である．**特権命令**とは普通のプログラムでは実行できない命令をいう．この命令はオペレーティング・システムだけで実行できる．

これらの命令と割込み処理ルーチンによって，プログラマは，事象の任意の系列に CPU がどのように対応するべきかを一意的かつ明確に指定することができる．割込み処理の詳細は本章の範囲を越えるので述べない．本節で示すのは単純な I/O 動作の処理概要である．

IBM 370 には次の5種類の割込みがある：

1. I/O
2. プログラム誤り
    a. 算術演算
    b. 不正命令
    c. 記憶侵害
3. スーパバイザ呼出し
4. 外　部
    a. タイマ
    b. コンソール割込みボタン
    c. CPU－CPU 割込み
5. 機械検査――機械の誤動作の検出

5種の割込み――I/O，プログラム誤り，スーパバイザ呼出し，外部，および機械検査――の各々に対して，“旧”と“新”の2つの PSW が計算機の記憶装置の中に格納されている．これらはあらかじめ定められた主記憶装置の番地に置かれる．割込みが起こると，割込みのハードウェア機構は現在の PSW を“旧”の場所に格納し，“新”の場所の PSW をロードする．次に CPU は PSW（“新”の PSW が入っている）の番地部によって指定された命令の実行を開始する．割込みルーチンの終了時には旧 PSW を現用の PSW にする命令が置かれていて，システムは以前の状態に復元され，割込まれたプログラム

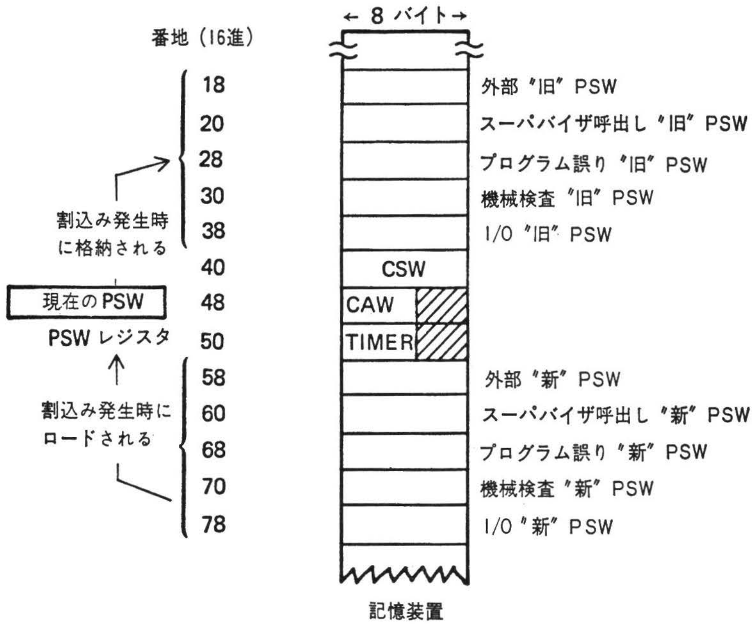


図 9.8　割込み用特殊番地

の実行が続行される．これらの特別番地は**図 9.8** に示されている．

割込みルーチンは，割込みを起こした条件を調べるために旧 PSW を参照することができる．旧 PSW は割込みコードと，割込みが起こったときに実行していたプログラムの番地を納めている．

割込みは，つねに，1つの命令が終了した後で，かつ，次の命令の始まる前に起こる．たいていの計算機システムにはマスクと呼ばれるハードウェアの機構を持つ．**マスク**は CPU の割込みを一時的に禁止する機能である．割込みは CPU がその割込み原因によって割込み可のときにのみ起こる．システム・マスク，プログラム・マスク，および機械検査マスクは割込みをマスクする機構である．割込みを処理中は他の割込みが起こると割込み行列ルーチンが呼出される．割込み行列ルーチンを実行している間に他の割込みが起こると，割込み行列ルーチンが再び呼出され，割込みが1つ失われることになる．したがっ

て，CPU が割込み行列ルーチンを実行中，他のすべての割込みはマスクしておかなければならない．

9.1.7 I/O 割込み処理の例（図 9.9）

前の3つの節では I/O 動作の3つの部分を調べた：

1. チャネルによる I/O コマンドの実行．
2. CPU による I/O 命令の実行．
3. チャネルと CPU の通信．これは割込み，CSW への情報の格納，または条件コード CC の設定で行なえる．

詳細をよりわかりやすくするために，簡単なI/O動作を最初から最後まで完全に見てみよう．

例は次のように構成されている：

1. CPU で実行される主プログラム——主プログラムは実行中に出力バッファの中に25行分のデータを作る．各行の長さは100字である．次に，I/O バッファの内容を全部2行ずつスペースをあけて25行印刷するために，I/O 命令を起動する．
2. チャネル・プログラム——チャネルはバッファの中の情報を上述のように印刷する．
3. 割込みルーチン——I/O チャネルが I/O プログラムを完了したとき起こす I/O 割込みの処理をする．割込みプログラムは CPU によって実行される．

この例は，チャネルおよび装置番号が X′00E′ の1403型印刷機で見出しをつけてページを印刷するプログラムの一部である．ここでは1ページ分を印刷するのに関係する部分だけを調べてみる．

START I/O 命令を実行する前にプログラマは次の用意をしておかなければならない：

1. チャネル・プログラムを作る．
2. データおよび定数領域を用意する．
3. バッファの中にページの内容を記入する．
4. CAW にチャネル・プログラムの番地を格納する．

ステップ番号	内容	番地（16進）	内容	註釈
10,13	I/O旧PSW	38	X'0000000EXX001500'	装置；命令番地
8	CSW	40	X'000021982C000000'	コマンド番地；状態；残り回数 = 0
2	CAW	48	002000	
		⋮		
11	I/O新PSW	78	X'XXXXXXXXXX000500	I/O割込み処理ルーチンの番地
		⋮		
12	割込みハンドラ	500	I/O割込み処理ルーチン	
		⋮	⋮	
		600		
		⋮		
(ここから実行開始)	主プログラム	842	次のページのために バッファに内容を作るプログラム	
		⋮	第1 CCMの番地をCAWに格納する	
1,2		1000	SIO　X'00E'	
		⋮		
3 9,14		1500	プログラム実行を続け，次の出力バッファに記入	
4	チャネル・プログラム	2000	CCW1　CCW　X'8B',*,X'40',0（次ページへ送る）	
5		2008	CCW2　CCW　X'19',PAGEHDR,X'40',11（印刷後3行送る）	
6		2010	CCW3　CCW　X'11',LINE1,X'40',100（印刷後2行送る）（CCWが23回続く）	
		⋮	⋮	
7		2190	CCW27　CCW　X'11',LINE1+(24*100),0,100	
		⋮		
	データ定数領域	2400	PAGEHDR　DC　C'PAGE HEADER'	
		⋮		
		3000	LINE1　DS　25CL100（印刷すべき25行の内容）	
		39C4	LINE2　DS　25CL100（第2バッファ）	

図9.9　I/O割込み処理が行なわれるプログラムの例

5.　スーパバイザ状態になる（PSW のビットを設定する）.

これらのステップを完了するとプログラマは START I/O (SIO) 命令を出すことができる.

註：CCW を書くために次のような新しい書式を用いる：

CCW 命令コード，番地，フラグ，回数

これはアセンブラの疑似命令であり，コアの2語境界に機械語の CCW を作る.

読者は次のステップ1から14までをたどってほしい.

CPU：

1. 装置 00E に START I/O が出される. CPU はチャネルと装置の状態を検査する.
2. CPU は, チャネルがチャネル・プログラムの番地(ここでは2000)を CAW から求めるきっかけを作る. チャネルはプログラムを実行し始める.
3. CPU は1004番地から再び実行を行なう. 一方チャネルは CCW1 の実行を開始し, 他の I/O 要求には動作中となる.

チャネル：

4. チャネルは CCW1 を解釈する.

   ビット 0-7　B′10001011′— 改ページする(次のページの先頭).

   32-35 B′0100′— コマンド・チェイン(次の CCW に行く).

5. チャネルは CCW2 を解釈する.

   ビット 0-7　B′00011001′—— 印刷し, 印刷後3行送る.

   8-31　X′002400′—— ページの見出し文字の格納番地.

   32-35 B′0100′—— コマンド・チェイン.

   48-63 X′000B′—— 11文字を指定する.

6. チャネルは CCW3 を解釈する.

   ビット 0-7　B′00010001′—— 印刷し, 印刷後2行送る.

   8-31　X′003000′—— 最初の行のデータの番地.

   32-35 B′0100′—— コマンド・チェイン.

   48-63 X′0064′—— 100文字を指定する.

   ⋮

   CCW4～CCW26 はステップ6と同様処理される.

7. チャネルは CCW27 を解釈する.

   ビット 0-7　B′00010001′—— 印刷し, 印刷後2行送る.

   8-31　X′003960′—— 最後の行のデータの番地.

   32-35 B′0000′—— これはチャネルによって参照される

最後のコマンドである.

48-63 X′0064′——　　100文字を指定する.

8. チャネルはI/O割込みを起こし，回数，装置番号，チャネル終了条件をCSWに設定して正常終了を通知する.
9. チャネルに対するマスク・ビット（現用PSWのビット0）=0であると，次の命令を実行する直前(たとえば，1500番地)に，CPUはI/O割込みを受理する（他に割込みが保留されていないとする).

CPU:

10. 現用のPSWは，装置番号と次に実行すべき命令の番地と共に38番地に格納される.
11. 現用のPSWは78番地にある新PSWで置換えられ，制御はI/O割込み処理ルーチンに渡される.
12. I/O割込み処理ルーチンはCSWを調べて動作が正常に終了したと判定する. このルーチンはあらかじめ定めておいた方法で元のプログラムにI/O動作の正常終了を通知する.
13. I/O割込みルーチンは現用PSWを旧I/O PSWの内容で置換えて主プログラムへ制御を返却する.
14. プログラムは割込みが起こる前に達していた所(1500番地)から続行する.

いまやチャネルは他のページの印刷を行なえる状態となっている. プログラマが2つのチャネル・プログラムと2つのデータ領域を持っていれば，第1ページが印刷中に第2ページの用意をすることができる. 第1のチャネル・プログラムが完了した後で(割込みによって通知される)，主プログラムは第2のチャネル・プログラムのためにCAWを指定してSIO命令を実行する. もし第1のチャネル・プログラムがページの印刷を完了していないときは，主プログラムは待ち状態にならなければならない. 第1のチャネル・プログラムが終了したとき(割込みを送る)，主プログラムが待ち状態であれば，I/O処理ルーチンは主プログラムのための旧PSWの状態をアクティブに変化させてから復元する.

9.1.8 多重プロセッサ

IBM 360 および 370 のチャネルあるいは I/O プロセッサは本来比較的単純なものである．特に，例外を除けば条件分岐を行なうことはできない．たとえば，これらのチャネルはカードを１枚ずつ読込んで“//JOB”というカードを読込んだら停止する（CPU 割込みを発生する）ということはできない．

CDC 7600 などの非常に大規模な計算機システムの I/O プロセッサには算術演算や条件判定命令などを持った小規模な中央処理装置やミニコンピュータのレベルにまで達しているものもある．さらに，処理量や信頼性の向上などのために，計算機会社は１つの記憶装置に何台もの中央処理装置の接続した大規模なシステムを製造している．このような一般的な多重プロセッサ・システムの管理は複雑であり，オペレーティング・システムによって制御する必要がある．

第 2 部

9.2 記 憶 管 理

記憶管理はオペレーティング・システムで重要な役割りを果たしている．記憶装置の利用はシステムの“あい路”となっていることが指摘されている．

前節でも述べたように，複数のプロセッサ（たとえば，複数の CPU と複数のチャネル）の最適な整合は複雑な問題である．記憶管理によってどのようにプロセッサの整合と仕事量の増加が行なわれるかを説明するために，本節では多重プログラミングの概念を紹介する．多重プログラミングについてはプロセッサ管理の節でより詳細に論じる．

本節では記憶管理の各種の方式について，必ずしも完全でないが広い範囲にわたって述べよう．より仕事量を増大させようとして行なわれたこれらの方式を簡単なものから複雑なものへの順で述べる．これらの技法の多くは説明している順番に発展したのでもないし，その所期の目的が単に効率の改善のみを目指したものでもなかった．しかしこれらの方式の歴史を詳細に述べることは本節の目的ではない．

本節に含まれている記憶管理方式は次の各種のハードウェア方式を使用して

いる：

1. 単一連続割付け
2. 分割割付け
3. 再配置型分割割付け
4. ページ割付け
5. デマンド・ページ割付け
6. セグメント割付け
7. セグメントーページ割付け

これらのハードウェアの機能を用いると記憶装置という資源の管理は非常に柔軟に行なうことができる．

9.2.1 単一連続割付け

単一連続割付け方式は**図 9.10** に示されている．

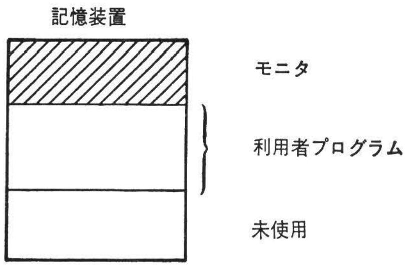


図 9.10　単純なバッチ・モニタ

オペレーティング・システム（または常駐のモニタ・システム）は，通常，主記憶装置の始めか終りに入っている．ジョブは，普通ごく一部の領域しか必要としないが主記憶の全部を割当てられる．ジョブは，終了するか誤りが起こるまでは CPU の制御を自分だけで行なっている．この方式は，特に，オペレータがテープやカード・デックを機械に掛けたりしなければならないようなシステムでは無駄が多い．I/O チャネルを持っているシステムでも，I/O チャネルが入力や出力を行なっている間 CPU がアイドルになっている時間は相当な割合にのぼる．

この型のシステムの大きな問題点は，資源が通常効率よく管理されていない

ことにある．システムには効果的な記憶割付けができるような柔軟なハードウェアが備わっていない．

この型の割付けは小規模で安価な計算システムによく用いられる．欠点はあるにせよ，大規模な計算機システムにおいてもプログラムの複雑さを最小とするために，この単一連続割付けシステムの単純性が大きく評価されることがある．

9.2.2 分割割付け

時間と記憶装置の無駄の問題を解決するために，分割割付け方式が導入された．これは記憶装置を分割して割付けるもので，IBM オペレーティング・システム 360 MFT（固定タスク数の 多重プログラミング）および MVT（可変タスク数の多重プログラミング）で用いられている．記憶域割付けの例は**図 9.11** に示されている．記憶領域は分割され，各ジョブ（またはタスク）が主記憶の別の部分に割付けられる（**図 9.11a**）．

分割割付けを行なうオペレーティング・システにおいては多重プログラムや多重タスクの技法により仕事量の増大が達成される．多重プログラムは，最も単純にいうと，２つ以上のジョブを同時に実行させ，または実行状態にしておくことをいう．これは一時に１つのジョブだけが実行される単純な単一割付けシステムの**順次処理**と対比される．

通常の多重プログラム分割方式では，CPU 時間は各ジョブにタイム・スライスと優先権を基準にして割当てられる．**タイム・スライス**（たとえば100 ミリ秒といった特定の時間）の間オペレーティング・システムはジョブにプロセッサを割付けて，次の４つの条件のいずれか１つが起こるまで，プログラムを番地空間（分割）の中で実行させる．

1. ジョブが終了する．
2. 誤りが検出される．
3. I/O が要求される．
4. ジョブに割当てられたタイム・スライスを使い果たした．

これらの各場合に，プロセッサは次に優先順位の高いジョブに割当てられる．初めの２つの場合には，ジョブを記憶装置から追出すことが可能である．後の

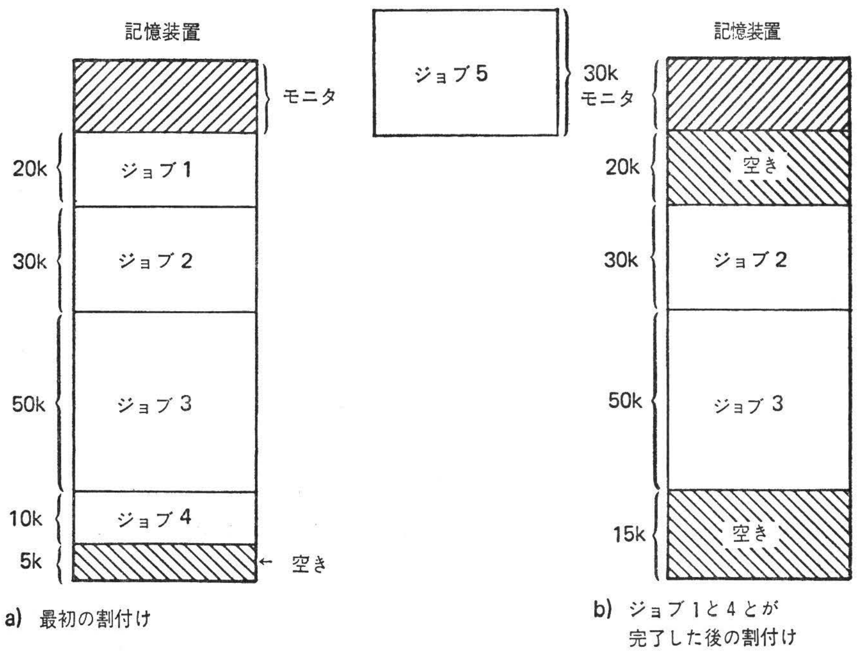


図 9.11 分割された記憶領域

2つの場合にはジョブは単に一時的に中断され，後で実行が再開される．

多重プログラミングの利点は上述した CPU のアイドル時間の無駄をなくしたり減少させたりできることである．たとえば，普通1時間かかる2つのジョブを多重プログラミング・システムで実行して両方共1時間で終了させるということも場合によっては可能である．これはもし各ジョブが別々に実行されるとI/Oや手操作の介入によって50%の CPU のアイドル時間が生じるような場合に起こる．バランスのとれた両方のジョブを同時に実行すると，ジョブ1が I/O を要求するときに，CPU はアイドルにならずジョブ2を実行できる（逆も同様である）．

ランダムにジョブを選んだときはジョブが完全にバランスすることはまれであろう．同時に実行するジョブの個数を増すと，すべてのジョブが同時に CPU を要求する確率が減少し，仕事量が増大する．通常のジョブの混合体では大体5つのジョブを同時に実行すると CPU の使用効率を約90%とするこ

とができる.

このような仕事量の増大には問題がないわけではない. 多重プログラミング・システムは次の理由により単純な単一割付けシステムよりも通常高価である:

1. 1つのジョブが誤りで他のジョブを壊すことを防ぐために, 特別の**保護機構**が必要である. 単一割付けシステムでは, ジョブは自分自身またはモニタを壊すだけであるからこれは不要であった.
2. 多重プログラミング・システムを完全に利用するためには, より複雑なオペレーティング・システムを必要とし, 主記憶に多数のジョブを入れるためにさらに余分の主記憶装置が必要となる.

分割割付け方式の大きな問題点は断片化である. **断片**とは使用できない"穴"すなわち"断片"が記憶装置の中で発達したものである. **図9.11**でこれを見ることができる. この例でジョブ1の番地空間は記憶域を20K, ジョブ2は30K, ジョブ3は50K, ジョブ4は10Kを占め, 5Kの未使用の記憶域がある. ("K"とは1000バイトの容量を示す). 記憶域を30K必要とするジョブ5が実行を待っていたとする. このジョブが開始できるためには連続した30Kのブロックが空きとならなければならない. 時間が経ってジョブ1が完了して記憶域を20K解放し, 次にジョブ4が完了して記憶域を10K解放したとする. したがって合計35Kの未使用記憶域ができたわけであるが, 連続しては30Kの記憶域がない. このためジョブ5はまだ実行開始できない. この種の分割方式では未使用領域の断片がしばしば発達する.

このようにして, たとえば (**図9.11a**のように) 4～5ジョブの多重プログラミングを想定している計算機構成において, 結果的には(**図9.11b**のように) ただの2～3ジョブが多重プログラミングされるに至る. この場合にはCPUの利用率と仕事量は減少する.

断片の問題は幾とおりかの方法によって解決できる. ここではそれらの内の2つ, 再配置型分割方式とページング方式を論じる.

9.2.3 再配置型分割割付け

断片をなくするために次の3つの方法を考えてみよう:

1. 何もせずに, 記憶装置を無駄にしても, CPUの使用効率が低くても満

足する．

2. 完全な多重プログラミングができる程度に主記憶装置の容量を増して CPU の使用効率を高める．

3. ハードウェアの機能を用意して断片をなくする．

最初の2つの方法は記憶装置に大きな無駄を生じる．今日の計算機システムの記憶装置はシステムの経費の 1/3 以上を占めることもしばしばである．したがって比較的安価なプロセッサの機能を用いて断片を解決しようとする幾つかの試みが行なわれている．

再配置型分割方式は断片の問題を解決する1つの方法である．これを**図 9.12** に示す．

この例では再び記憶装置の中に4つのジョブを考える（**図 9.12a**）．断片のために新しいジョブがロードできないときには，主記憶中のジョブは上の方へ詰

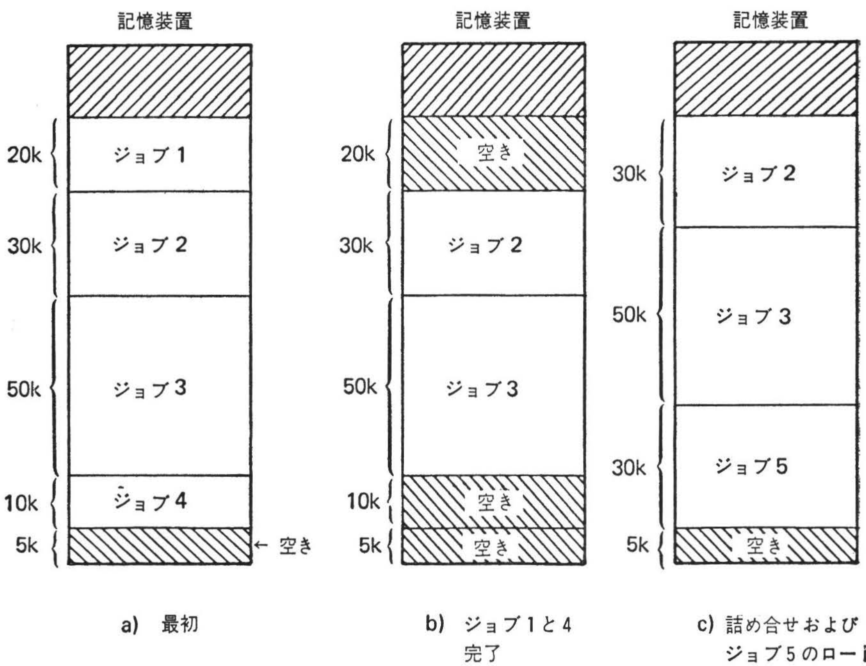


図 9.12　再配置を行なう記憶領域の分割

め合わされる．たとえば，ジョブ1およびジョブ4が終了すると（**図9.12b**），ジョブ2とジョブ3は移動され，こうして生れた一続きの領域にジョブ5を入れることができる（**図9.12c**）．

分割部分は特別のハードウェア機能がないと効率よく再配置することができない．いったんロードされるとプログラムには植えつけた種子のように“根”が生えてくる．“根”は次のような番地に関係するプログラムの要素によって発達する：

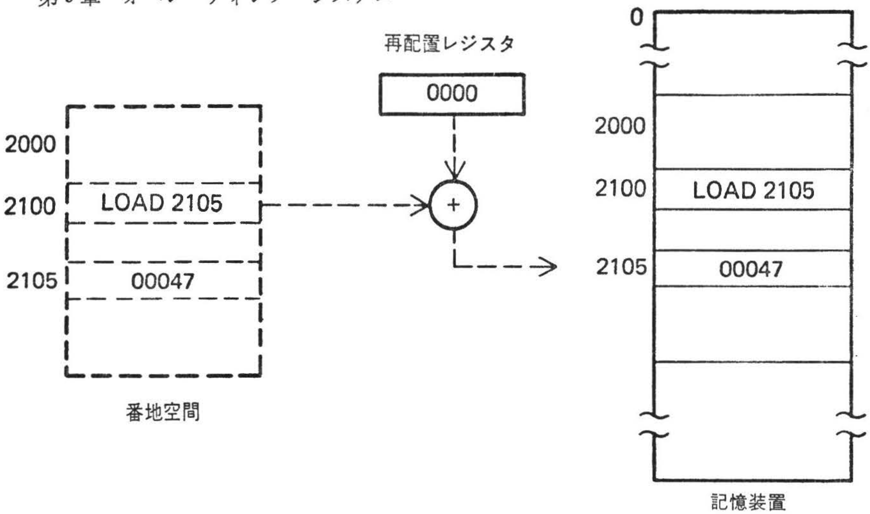
1. 記憶装置を参照する命令で“番地ポインタ”を使用するもの．
2. 引数表で“番地ポインタ”を使用するもの．
3. データ構造で“番地ポインタ”を使用するもの．

したがって，分割部分の内容を単に別の場所に移しただけでは番地に関係する要素が適当に修正されないので，そのプログラムは一般には新しい場所で正しく働かないことになる．困ったことに，たいていの計算機ではデータと番地とを（“黒”はデータであるとか，“赤”は番地であるというように）区別する方法がない．したがって，モニタにはどの語を変更しなければならないかが正確にはわからない．このために使用できるソフトウェアの技法はあるにはあるが，非常に効率が悪いか，または，制限が多い（第10章参考文献を参照）．

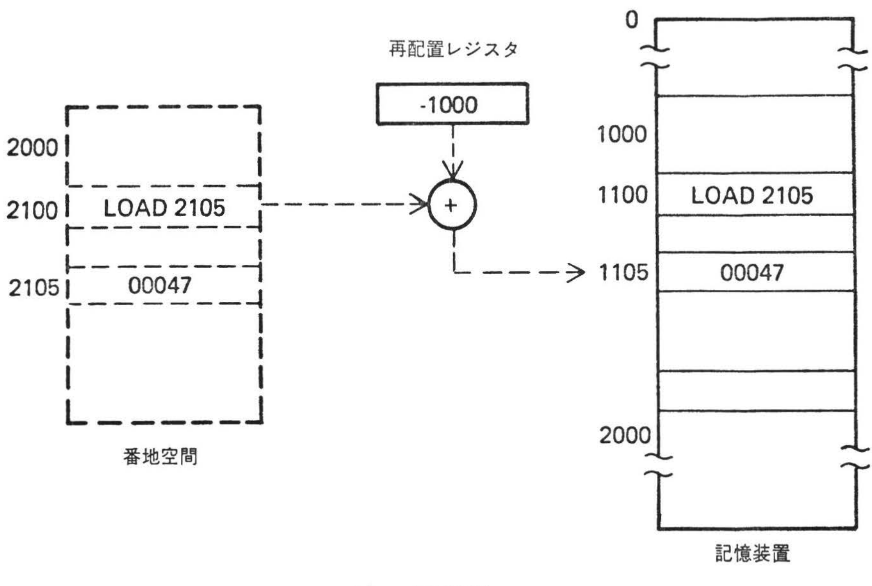
この問題の解決方法として，ロードし直して最初から実行し直す方法もある．これは非常に経費がかさむうえに，プログラムがすでにマスタ・テープを更新したり，変更の最中であったり，非可逆的な処理を実行していたりする場合には適用できない．

プログラムが論理的な記憶場所に独立となるような“相対性”の機構が必要である．これは物理学の相対性と類似している．“理想的な”車の中に閉じ込められた人はどちらに向って動いているか，または，停っているか否かさえもわからない．この目的のために用いられる簡単な機構は**再配置レジスタ**である．このレジスタの内容は記憶装置を参照するのに使用されるすべての番地に自動的に加算される．この再配置レジスタは記憶装置を参照するすべての命令に対して自動的に使用される“隠された”インデックス・レジスタと考えることができる．

図9.13の例で，1つのジョブの番地空間（左）は，物理的な記憶装置の配



a)　再配置前



b)　再配置後

図 9.13　再配置レジスタの使用

置（右）から分離されている．再配置の前には，2100 番地の命令 LOAD 2105（2105 番地からアキュムレータに LOAD する）[1] は計算機のアキュムレータに値 00047 をロードする．再配置の後ではプログラムは 2000 番地でなく 1000 番地から始まる．しかしすべての命令には何も変更がない．プログラムは 2000 番地から 1000 番地へ移されているので，ハードウェアの再配置レジスタはモニタによって 1000−2000=−1000 と設定されている．したがって，命令 LOAD 2105 が出てくると，物理的な主記憶の番地 2105 からアキュムレータへロードせずに，番地 2105−1000=1105 の内容がロードされる．このようにして値 00047 の参照が行なわれる．重要なのは，プログラムの中では実際の物理的な主記憶装置の番地を知らないことである．この例ではすべての LOAD, ADD, BRANCH などの命令は物理的には 1000 番地に移されていても，2000 番地にあるかのごとく実行される．

再配置型割付けを行なう計算機システムでは特定の物理的な番地とプログラムの番地とは無関係であるので，通常すべてのプログラムは 0 番地からロードされたとして処理される．このようにすると再配置レジスタの内容はつねに正である．

再配置の機構は全く単純であるが，概念的には重要な意味を持っている．前の例で示したように，ジョブの番地空間（名前空間）はプログラムの物理的な主記憶装置の番地と独立である．この独立性の概念は後に述べる複雑な記憶割付け方式の基礎となる．

この方式には主として 2 つの問題がある．第 1 は再配置のオーバヘッドである．ジョブが終了したとき，システムは再詰め合わせのために他のすべてのジョブを再配置しなければならないこともある．分割方式と再配置分割方式とは，CPU 時間（再配置型分割方式のオーバヘッドである再詰め合わせ）と記憶装置の無駄（分割方式での断片）とを比べて決定しなければならない．たとえば Honeywell 635 計算機システムは再配置型分割方式を用いている．第 2 の問題は依然として無駄になっている記憶域が存在することである（**図 9.12a** 参照）．

9.2.4 ページ割付け

1　この例では 1 番地方式で語形式の計算機を仮定する．

ページングは断片の問題の他の解決法である．ページング方式は XDS 940 など幾つかの計算機で使用されている．**図 9.14** に示された例を考えてみよう．

図でジョブの番地空間は1000バイト単位の**ページ**に分割されていると考える．再配置型分割方式と同様に，分割は番地空間に何も影響を与えないので，利用者にはこの分割はわからない．物理的な記憶装置も1000バイト単位の**ブロック**に分割されている．ページとブロックは差し込みの2つの部分のように考えることができる．すなわち，ページは任意のブロックに“差込む”ことが

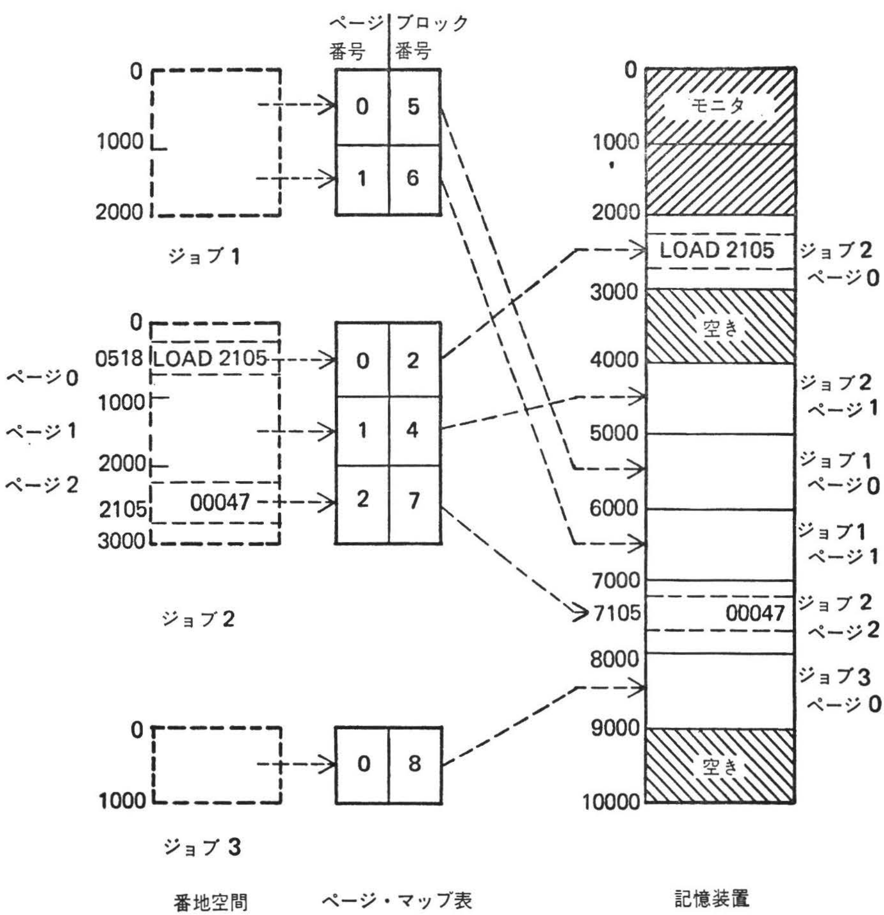


図 9.14　ページング方式による記憶割付け

できる．再配置型分割方式と同様に，利用者の番地空間から物理的な記憶装置への写像を行なうハードウェアの機構を用意することが必要である．この目的のために，各ページに対して別々のレジスタが置かれている．これらはしばしば**ページ・マップ**とか**ページ・マップ表**と呼ばれる．これらのレジスタとしては特別のハードウェア・レジスタが使用されたり主記憶装置の一部が使用されたりする．ページ・マップ表は番地空間そのものよりもはるかに容量が小さいことに注意してほしい．**図 9.14** ではこれらの大きさが比例して書かれていない．たとえば，ジョブ２の番地空間は 3000 語であり，ページ・マップ表はただの６語である．

各ページは別々に再配置することができるので，１つのジョブに対する記憶域は記憶装置の中で完全に連続している必要がない．１つのページの中でだけ連続であればよい．明らかに，ページの大きさの選定はこの方式の効率に大きな影響を与える．もしページが大き過ぎるとジョブの分割の大きさに近くなり，再配置型分割方式となる．ページが小さ過ぎると大きいページ・マップが必要となり，計算機システムの経費を増大させる．これらの理由などからページングを行なうシステムの多くは大体 4000 バイトのページを採用している．

図 9.14 の例のシステムでは，ジョブ１の番地空間は２ページに，ジョブ２は３ページに分けられ，ジョブ３は１ページに入る．各ジョブごとにページ表が存在し，この表にページ番号とそのページの物理的な記憶番地（ブロック番号）が記入されている．各番地に対しては写像が存在する．すなわち，ジョブの番地空間は物理的な記憶装置の中の番地に変換することができる．

番地空間の 0518 番地に示されている LOAD 2105 命令はページ０の 518 番地に置かれている．同様に 2105 番地のデータ語 00047 はページ２の中の 105 番地に置かれていると考えられる．ページ表は番地空間のページ番号を物理的な記憶装置のブロック番号に写像する．このようにして，LOAD 命令は，ページ表がページ０はブロック２に格納されていることを示しているので，物理的な 2518 番地（ブロック２の中の 518 番地）に置かれる．ハードウェアはこの番地の左側の何桁かを取り出してページ表の指標として使用し，この部分をページ表の中の値で置換えて番地の写像を行なう．ハードウェアの方式によっては，番地の写像を行なう CPU 機構は計算機の速度を多少遅くする．

ページングの方式を用いると，断片の問題をジョブの終了の後で分割をやり直さずに解決することができる．たとえば，**図 9.14** では2000語の未使用の記憶域がある．しかしこれらは連続していない．第4のジョブで番地空間が2000語以内のものがあれば，領域を再配置し直して2000語の連続した部分を作ることもできる．しかし領域を再配置せずに，ジョブ4の番地空間を2ページに分割し，1つを3000番地に他を9000番地に置くこともできる．次に示すようにページ・マップを設定しておくと，番地空間はあたかも連続しているかのように見える：

ページ	ブロック
0	3
1	9

利用者4のページ・マップ表

ページングに関しては幾つかの問題点がある．第1は各番地変換に関連したオーバヘッドである．他の問題はページ表のために必要となる余分の主記憶または余分のレジスタである．ジョブの長さは1000バイトの倍数でないので最後のページの一部分は無駄となる．これは**ページ余り**と呼ばれる．平均的には各ジョブについて半ページが無駄になっている．これは小さなジョブが多数存在するとき重大な問題となる．

さらに3つの欠点がある．第1，再配置型分割割付け方式と同様に，あるジョブをすべて入れるには小さすぎるという場合には依然として主記憶に無駄ができることである．第2，この種のページング方式では，プログラムが実行される前にそのジョブのすべてのページが主記憶に入っていなければならないことである．第3，かなりの程度のハードウェアによる援助が必要である．

9.2.5　デマンド・ページ割付け

デマンド・ページングは，（ほとんど）無限の記憶容量を与えるというプログラマの夢を満たす記憶割付け方式である．デマンド・ページング方式はIBM 360 モデル67，Honeywell 645 および RCA 3 と RCA 7 で使用されている．

仮想記憶とは，デマンド・ページングでジョブが働く番地空間の別の名前である．仮想記憶は，原理的に，使用できる記憶域を大きくすることができる．

デマンド・ページングの記憶割付け方式は**図 9.15** に示されている.

図 9.15 においてはジョブの番地空間は 10,000 バイトのプログラムとデータから構成され，使用できる記憶装置はただの 3000 バイトであると仮定している．この 10,000 バイトのプログラムはデマンド・ページングによって記憶容量が 3000 バイトの計算機で実行できる．ページ表（**図 9.15**）は各ページの欄に幾つかの部分――ページ番地，ブロック番号（主記憶中の番地），状態（主記憶の中にあるか外にあるか），および後述する判定情報――を持っている.

割付けアルゴリズム（ソフトウェア）は，プログラムの 3 ページを主記憶に入れることができる．今最初の 3 ページが主記憶に入れられたとして，ジョブはこれらのページの 1 つの中で実行を開始するとする．ページ 0 の中で 9005

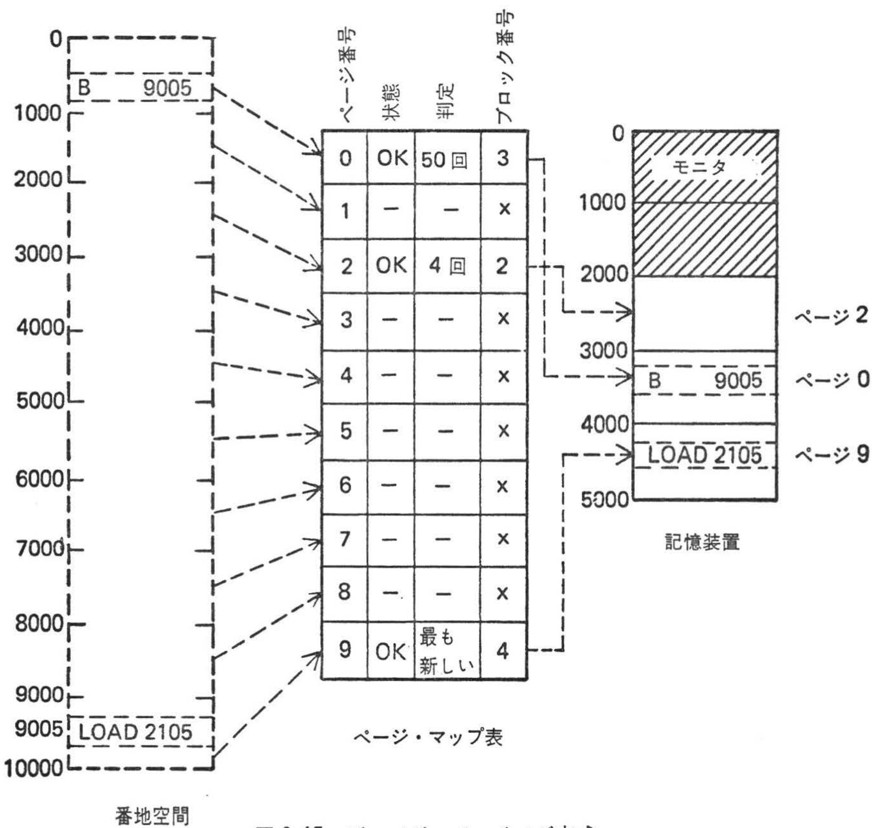


図 9.15 デマンド・ページング方式

番地へ制御を移す命令があると，ハードウェア機構は9をページ表の指標として用い，そのページの状態を検査してこれが主記憶の中にないことを知る（すなわち，"OK" でない）．そこでページ9を主記憶の中に入れるように**要求**(demand) する．ソフトウェアは2次記憶を探してページ9を求め，3ページの内の1つの上に重ねて書き込んで（すなわち，置換え）それをロードする．どのページを置換えるかを決定するために判定ビットが使用される．理想的な置換え方法は**最も使われない**ページを置換えることである．多くのプログラムではプログラムの実行は全番地空間にわたって一様に分布しているのではなく，プロセッサが多くの時間を費やす局所的な番地の部分が存在する．よく使用されるページだけを主記憶に保っておくことが望ましい．どのページが主記憶に留まり，どのページは留まるべきでないかを決定する**ページング・アルゴリズム**が多数存在する．1つのページング・アルゴリズムは単にこのページに対する参照回数を判定部に入れるものである．置換えられるべきページは参照回数が最も少ないものである．当然最も新しく主記憶に入れられたページは最も回数が少ないから，この判定部は定期的にリセットしなければならない．最も新しいページは置換えられるべきではない．

デマンド・ページングを実用化しているシステムの多くは "デマンド" 処理の一部分だけをハードウェアで行なっている．ページの状態が "主記憶に入っていない" （OKでない）ことをハードウェアが見つけると，処理を中断し，自動的に割込みを発生させてモニタに制御を移す．この割込みはしばしば**プロセッサ割込み**または**ページ割込み**と呼ばれる．ここでモニタは実際に2次記憶装置を参照して必要なページを求める．さらに，ページが主記憶に転送されている間，モニタは前に述べたページングを行なっているシステムと同様に，他のジョブを多重プログラムで実行することができる．

非常に大きい記憶装置を使用できるというプログラミング上の利点に加えて，デマンド・ページングは幾つかの面でシステムの仕事量を増大させることができる．研究の結果ではジョブの番地空間は実際には一部分だけしか使用されないことが示されている．これは誤り処理ルーチン——しばしば実行中に使用されない——や実行中のデータによってごく一部分だけが使用される表などを考えてみればよくわかる．このことはこれらの部分がプログラムに不要だと

いうではなく，これらの部分は実行中に必要になるまでロードしない方がよいということを意味している．典型的なプログラムの番地空間の25%に達する部分が実行中に使用されないという報告もある．したがって，デマンド・ページングを用いると，特別な工夫をしなくてもシステムは25%多い利用者を多重プログラムで実行することができる．

さらに，初期化ルーチンや多くのプログラムにおいて‘順番に実行される’段階（たとえば，コンパイラのサブルーチン）など，プログラムの多くの部分はただの１度しか使用されない．この特性はプログラムの性質に大きく依存するけれども，短い時間の間ではプログラムの全番地空間の50%よりはるかに小さい部分だけが必要であるということが普通である．ある時間内に使用される部分を**ワーキング・セット**という（これらの概念については第10章の参考文献に詳しく論じられている）．

多数のプログラムを同時に実行する“タイム・シェアリング・システム”では比較的短い時間間隔を用いるのでデマンド・ページングは非常に有効である．本章の初めに述べたように，通常５〜６ジョブの多重プログラミングで最適化した仕事量に近くなる．タイム・シェアリングの場合，多数のジョブを同時に実行すると，TSSでは重要な応答時間が改善されるが，反面仕事量は減少する．もちろん極端に多くのプログラムがタイム・シェアリングされると，主記憶装置の容量が小さいシステムでデマンド・ページングを行なったとき（またはまずいページ判定アルゴリズムを用いたとき）と同様に，ページが主記憶に読込まれ，直ちに置換えられ，また読込まれるということが絶えず行なわれる結果仕事量が極端に減少することがある（これらの場合には仕事量は１%以下になることが知られている）．このような極端な場合，システムには**スラッシング**（thrashing）が起こっているといわれる．デマンド・ページング方式は重要であり，スラッシングの問題は複雑であるから，スラッシングの確率を最小にするためのアルゴリズムを開発する多くの研究が行なわれている（第10章参考文献参照）．デマンド・ページングも前節のページングで述べた欠点を持つ．

9.2.6 セグメント割付け

セグメンテーションは記憶装置と情報の両方の管理方式である．最初記憶管理の面に焦点を合わせることにする．第9.5節では情報管理の面に焦点を合わせることにする．再配置型分割方式，ページング方式およびデマンド・ページング方式のオペレーティング・システムでは，ジョブの番地空間と物理的な主記憶装置との間の独立性の概念を用いてはいるものの，依然としてプログラマは番地空間を，単一の直線的な記憶装置であるかのように考えている．そこで，“プログラマにとって番地空間はどのように構成されているのがよいか”を考えてみて，その次にこの番地空間を再配置およびページングの機構と同様の技法を用いて組込んでみよう．

セグメントは1つの物として取り扱われる1かたまりの情報である．たとえば，1つのプログラムまたは複数のプログラム，1つのデータ・ベースまたは複数のデータ・ベースなどである．セグメント化されている場合のプログラムでは番地空間を2次元空間で，セグメントの名前とセグメント内のオフセットの2つの成分で番地指定される．各セグメントは固有のアクセス権を持っている．セグメンテーションによってダイナミック・リンキングの組込みが可能となる．ダイナミック・リンキング方式では CALL 文を実行するときに，ローディング機構がプログラムを探し，主記憶にロードし，呼出されたプログラムを呼出したプログラムに**実行時に**結合する．この結果もしプログラムが実際に呼出されなければそれに伴うオーバヘッドはない．

図9.16は，プログラマにはセグメント化された番地空間がどのように見えるかを示したものである．各セグメントは，独立の長さを持つ別個の直線的な記憶装置のように見える（すなわち，セグメントは同じ長さである必要はない）．図で ALPHA と BETA は実行可のプログラムを持つセグメント，DATA はデータのみを持つセグメントとする．2次元的な番地空間を持つセグメント化システムでは，すべての命令は番地参照を2つの部分——セグメント名とそのセグメント内でのオフセット（番地）——で表現しなければならない．たとえば，代表的なアセンブラ語の文は次のように書かれる：

LOAD 〈DATA〉|25　セグメント DATA の25番地から1語をロードする．

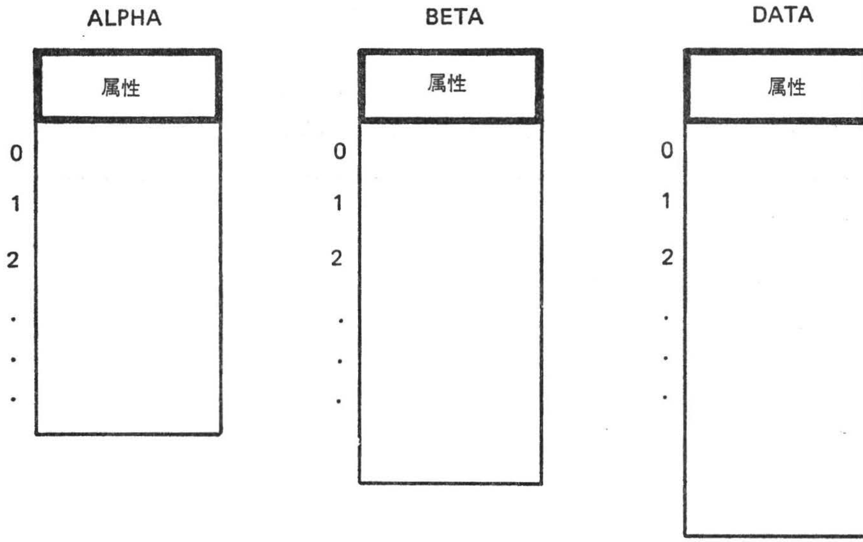


図 9.16 利用者から見たセグメント化した番地空間

TRANSFER ⟨BETA⟩|7　　セグメント BETA の中の7番地に制御を移す.

記号"|"は番地部のセグメント指定とオフセット指定を区切る.

セグメント化された番地空間を実際に組込む際には，ハードウェアとプログラミングを簡単にするために幾つかの技法が使用される．一般に，セグメントは名前ではなく番号で参照される．これはちょうどアセンブラ語のプログラムが記憶場所を記号で参照するが，機械語プログラムでは数値の番地を使用するのと同じである．したがって，次の記号による命令：

TRANSFER ⟨BETA⟩|LOOP+2

は，次のような形で実行されねばならない：

0500 3|26

ここで BETA のセグメント番号は 3，LOOP の BETA の内での番地は 24，0500 は TRANSFER 命令の機械語コードとする．多くの場合インデックス・レジスタや間接番地の機構は番地指定法を容易にするために使用できる.

プログラマから見たとき，セグメント化された番地空間には幾つかの利点がある．その1つとして読出し専用，書込み専用，実行専用などのアクセス権の特定の属性をセグメントごとに対応させることができる．実行専用と規定されたセグメントの中の語は読出したり書込んだりできない．したがって，ALPHAが実行専用と規定されていると，ハードウェアは次のような命令の実行を許可しない：

LOAD　　　〈ALPHA〉|22

または

STORE　　　〈ALPHA〉|12

しかし次は実行できる：

TRANSFER　〈ALPHA〉|0

この機能は事故によってプログラムが破壊されることを防ぎ，使用料が必要な“専用”サブルーチンの秘密を確保する．同様にして，読出し専用のセグメントは事故または故意の変更から守らなければならないデータの格納に使用することができる．

セグメントの長さが任意であるということは，プログラマがセグメントの最大長を前もって規定する必要がないことを暗に示している（註：組込みの制限から命令のセグメント・オフセット部が有限個のビットであるので，任意長ということは無限長を意味するのではない）．このように，セグメントは長さの予想ができない表や行列を構成するのに非常に都合のよい方法である．次の命令：

STORE　　　〈DATA〉|1500

は，セグメント DATA がまだ大きくなければ1500番地までセグメントの大きさを拡張する．

セグメント化された番地空間はプログラミングにとって大変便利であるが，オペレーティング・システムの資源管理にも利点がある．セグメント属性や自動的なセグメント長拡張機能を，オペレーティング・システム自身の機能を助けるために使用することができる．しかも，仕事量の大幅な増大がダイナミッ

クな結合やセグメントの共有を行なうことによって達成できる.

これまでのシステムでは，ジョブのすべてのサブルーチンは実行開始以前に個々のサブルーチンを再配置して直線的な番地空間の中に結合されていると仮定していた．第5章で論じた**リンケージ・エディット，バインディング**または**再配置ローディング**と呼ばれる処理は，ジョブを完了させるのに必要な時間のかなりの部分を占めた．多くの場合この処理はジョブのコンパイルや実行と同程度に時間がかかった．セグメント化したシステムではサブルーチン・セグメントの中で番地を再配置する必要はない．単に必要なことは，セグメント間の記号参照を数値のセグメント番号に変換することである．コンパイラが生成した文字列の形のままで，まだ結合されていないものと，実際に結合されたセグメント番号とを判別するハードウェアの検出機構を備えることによって，セグメント間結合はそれを必要とする命令が実行されようとしたときのみ行なわれる．これは**ダイナミック・バインディング**とか，**ダイナミック・リンキング**とか**ダイナミック・ローディング**とか呼ばれる．次のようなプログラム ALPHA[2]：

```
        ・
        ・
        ・
ALPHA|0201      LOAD        ⟨DATA⟩|5

ALPHA|0500      TRANSFER    ⟨BETA⟩|0
        ・
```

では，命令 LOAD ⟨DATA⟩|5 は，それが実行されようとしたとき結合処理が実行され，次のように変更される：

```
        ・
        ・
        ・
ALPHA|0201      LOAD        4|5
        ・
        ・
        ・
ALPHA|0500      TRANSFER    ⟨BETA⟩|0
```

2 割付けた番地の値を命令の中に直接入れることは ALPHA を純粋でない手続きにし，セグメントの共有の面から支障を生じる．これについては情報管理の節で論じる．

ここで DATA のセグメント番号4はオペレーティング・システムのダイミナック・バインディング部で割当てられたと考えている.

命令 TRANSFER 〈BETA〉|0 が条件によって，または，異状終了によって実行されなかったならば，この命令が行なう外部参照は結合されず，また BETA はロードすらされない．情報検索システムやコンパイラなどの大規模で複雑なプログラムの中で用いられる多数の“選択的”なサブルーチンは，特定の実行の間に必ずしもすべて要求されるわけではない．したがって，実際に必要なもののみを結合することによって結合の時間と記憶容量を節約し，計算機システムの仕事量を増すことが可能である．セグメンテーションを処理するためのハードウェアなしにダイナミック・リンキングを行なうソフトウェア技法が存在するけれども，これらは通常効率も悪いし種々の制限をも伴う.

前に述べた記憶管理方式は，与えられた記憶装置にジョブを“詰め込み”，できるだけたくさんのジョブを多重プログラミングで実行して仕事量を増加させようという動機から生れたものであった．セグメンテーションは**共有**によって多重プログラミングの仕事量を増大させる別の技法である.

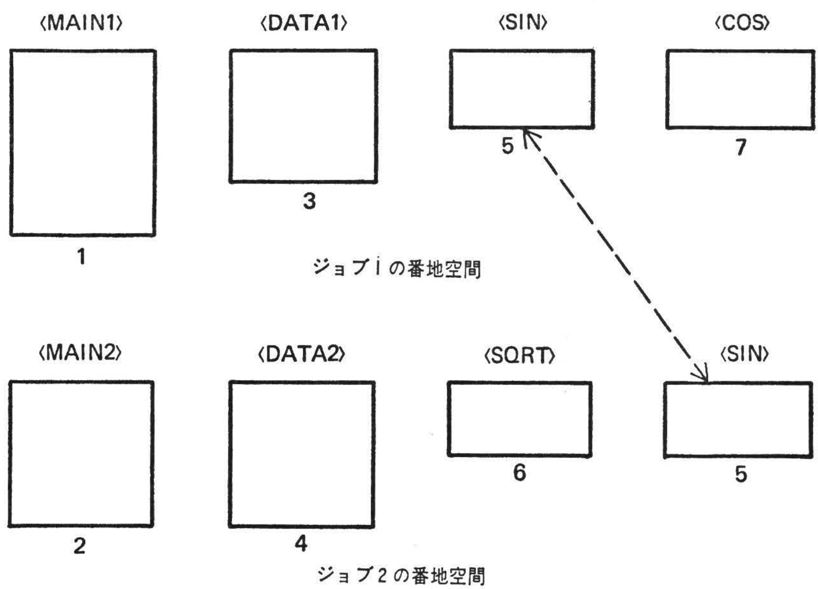


図 9.17　セグメント化された多重プログラミング・システムでの共有

図 9.17 では2つのジョブが実行されている単純な多重プログムミングの状態を示す．各ジョブは番地空間に4つのセグメントを持っているが，両者共同じシステム・ライブラリ・ルーチン SIN を必要としている．普通のシステムでは，各利用者は自分の番地空間に結合された別々の SIN ルーチンの写しを持つことになる．もし SIN ルーチンが**共有可能**であれば別個の2つの写しを持つ必要はない．

プログラマは普通**連続再実行可能** (serially reusable) なプログラムを書く．連続再実行可能ルーチンとは使用の度に初期化を行なって何回でも再使用可能なルーチンをいう．ルーチンは呼出される度に，前の呼出しで残された値を使用しないように，適当な初期化手続きを用いることによって連続再実行可能とすることができる．呼出される度に異なった値を返却することを目的とした**乱数発生ルーチン**など，ルーチンによってはこの条件をわざと無視しているものもある．多重プログラミング・システムにおいて効果的に共有が行なわれるためには，ルーチンは**同時に再実行可能** (concurrently reusable) でなければならない．これは**再入可能** (re-entrant) や**純** (pure) 手続き（自分自身を変更しないルーチン）ともいわれる．

連続再実行可能ルーチンは，他のジョブで使用される前につねに1つのジョブの実行が完了するとき共有することができる．この制御は行列またはインタロックを用いることによって行なうことができるが，複雑で効率も悪いので，ルーチンの共有を行なうシステムでは，普通，再入可能であることを要求している．

再入可能なルーチンは，実行が完了する前に多重呼出しと多重実行を許す．制限のつかない再入可能性を実現するためには，ルーチンは実行・読出し専用コードでなければならない．この純手続きはレジスタ，または，ジョブごとに別に作られたデータ・セグメントに置かれた変数だけを使用して動作する．そして自分自身に格納したり自分自身を変更したりしない．ダイナミック・リンキングの説明に用いた例は，ロード命令の変更が必要であったので純粋ではない．同じことを行なうが“純粋性”を損わないダイナミック・リンキングの技法が存在する．

図 9.17 では，SIN ルーチンが純手続きで，変化するデータ記憶域として

DATA1 と DATA2 を使用している．この2つのジョブは1つの SIN を共有することができる．参照はデータ・セグメント番号をインデックスまたはベース・レジスタに納めておくことによって行なうことができる．

ライブラリ・ルーチンのほかに，共有するとよいシステム要素がたくさんある．ファイル・システムや入力／出力ルーチンなどの各種のシステムの設備は共有できる．プログラマは互いにルーチンやデータ・セグメントを共有したいであろう．PL/I コンパイラや BASIC インタプリタなどのシステム全体も共有できる．このような場合，たとえば PL/I プログラムをコンパイルする10個のジョブが同時に実行されたとする． PL/I コンパイラは 100,000 バイトのプログラム・セグメントと，コンパイル中に使用する各種の表や変数のために 10,000 バイトのデータ・セグメントとが必要であるとすると，10ジョブのために必要な合計の記憶容量は 10×(100,000＋10,000)＝1,100,000 バイトである．1つの PL/I コンパイラ・セグメントを共有して，別々のデータ・セグメントを使用すると，実際に必要な記憶容量は 100,000＋10×10,000＝200,000 バイトに減少する（82％ の減少)．

これまでは，セグメント化した記憶装置が物理的に作れると仮定してきた．実際の記憶装置は直線的なものであるから，見掛けは別々のセグメント化された番地空間を直線的な記憶空間に変換する**セグメント・マップ**を使用する必要がある．これは，見掛けは直線的な番地空間を不連続なページとブロックに変換するために使用したページ・マップと同様の働きをする．**図 9.18** はこのようなセグメント・マップを示している．

各々のセグメントは記憶装置の連続した部分に割付けられている．この例では，ハードウェアのセグメント・マップ機構は自動的にセグメントの番地を物理的な番地に変換する．すなわち，セグメント番号をセグメント・マップの指標として用いて求めたセグメントの先頭の番地にセグメントのオフセットを加えて物理的な記憶装置の番地が求められる．また，セグメント・マップは現在の長さを示している部分を持っている．セグメントの長さを越えたオフセットを使用しようとすると，ハードウェアにより自動的に検出されて割込みが起こり，モニタによりセグメント長の拡張が行なわれる．

Burroughs 5500 などの計算機は上述のセグメンテーション・システムを用

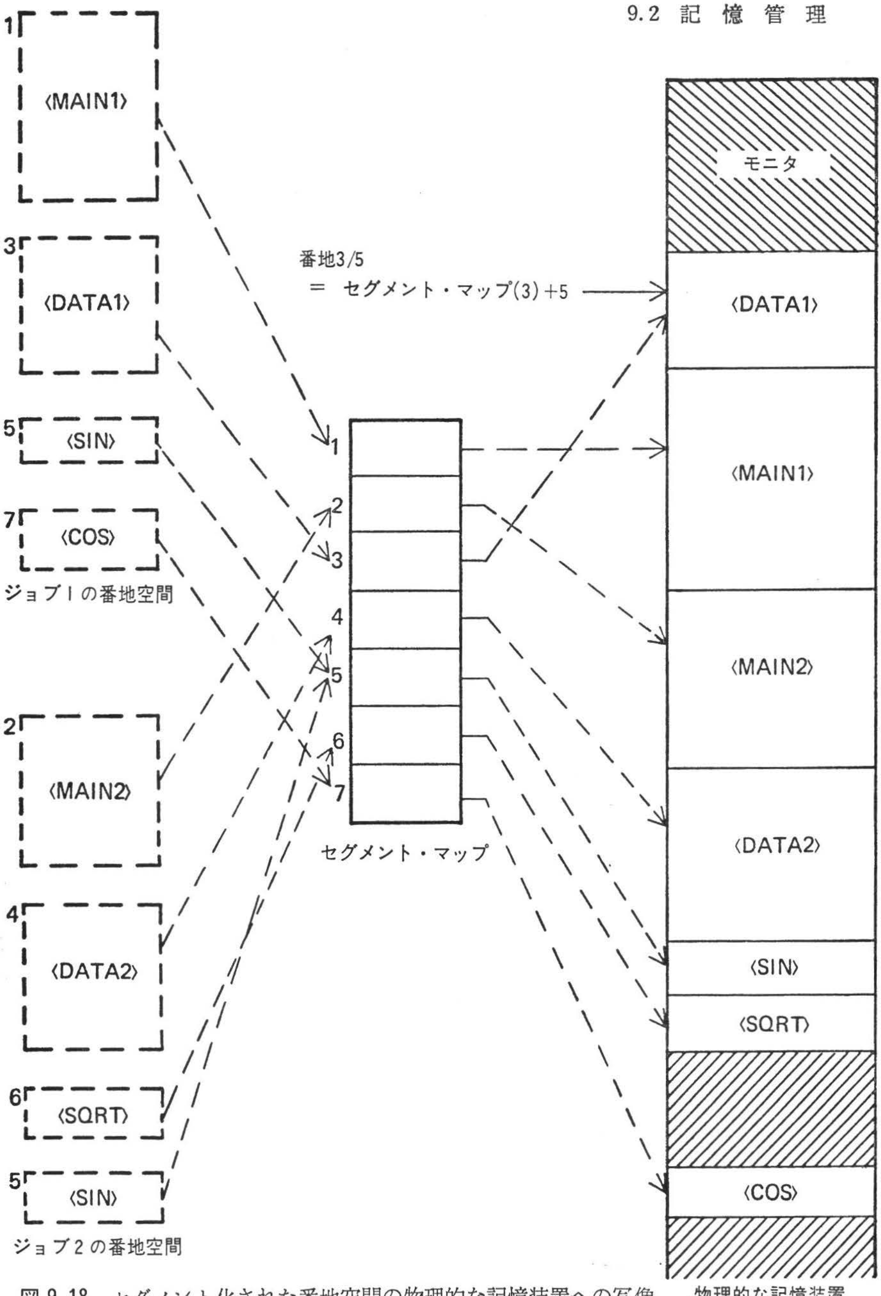


図 9.18 セグメント化された番地空間の物理的な記憶装置への写像

いているが，２つの大きな欠点がある．第１，各セグメント内の語が連続でなければならないので，特に，セグメントが動的に大きくなることを許される場合には断片が大きな問題となる．第２，番地空間の合計の容量は物理的な記憶装置の容量で制限される．ある意味では，セグメント化されたシステムは再配置型分割方式と同じ記憶装置割付けの問題を持つ．

9.2.7　セグメント-ページ割付け

そこで前述の技法の優れた点を組み合わせることを考える．セグメンテーションは情報の共有と保護を容易とし，ページングは断片と再圧縮の問題を解決した．またデマンド・ページングは番地空間の容量の制限を取り除いた．セグメンテーションとデマンド・ページングを組み合わせたこの方式は IBM 360 モデル 67 や HIS 645 などの幾つかの進歩した計算機システムで採用されている．**図 9.19** は各セグメントごとに別々のページ表を用いてセグメント化したシステムを示している．読者はこれを**図 9.18** と比較してほしい．

セグメンテーションについては第 9.5 節にて再び触れ，情報管理の役割りを調べるが，ここでは記憶管理に関する幾つかの点を指摘しておく．セグメント表やページ表はその容量の関係から，ハードウェアのレジスタでなくて主記憶装置に格納される．したがって，１つの命令を処理するとき，データ語をロードするためには記憶装置を３回参照しなければならない（１回目はセグメント表を，２回目は対応したページ表を，そして３回目に実際のデータの場所を参照するため）．頻繁に使用されるセグメントやページの先頭番地を小容量高速度の連想記憶バッファで“記憶する”ことは，必要となる記憶装置参照の回数を減少させるのに非常に効果がある．この方法は IBM 360 モデル 67 および HIS 645 計算機の両者で用いられている．

しかしながら，連想記憶装置を用いても，多層の番地指定方式を用いると計算機の速度が低下する．通常この速度低下は 10% 以下とすることができるが，プログラムの性質や連想記憶バッファの性能によって非常に大きく変化することがある．もちろん，セグメンテーションとページングのハードウェアは計算機システムの経費を増大させる．

セグメント－ページ方式の性能を完全に利用するためには，明らかにオペレ

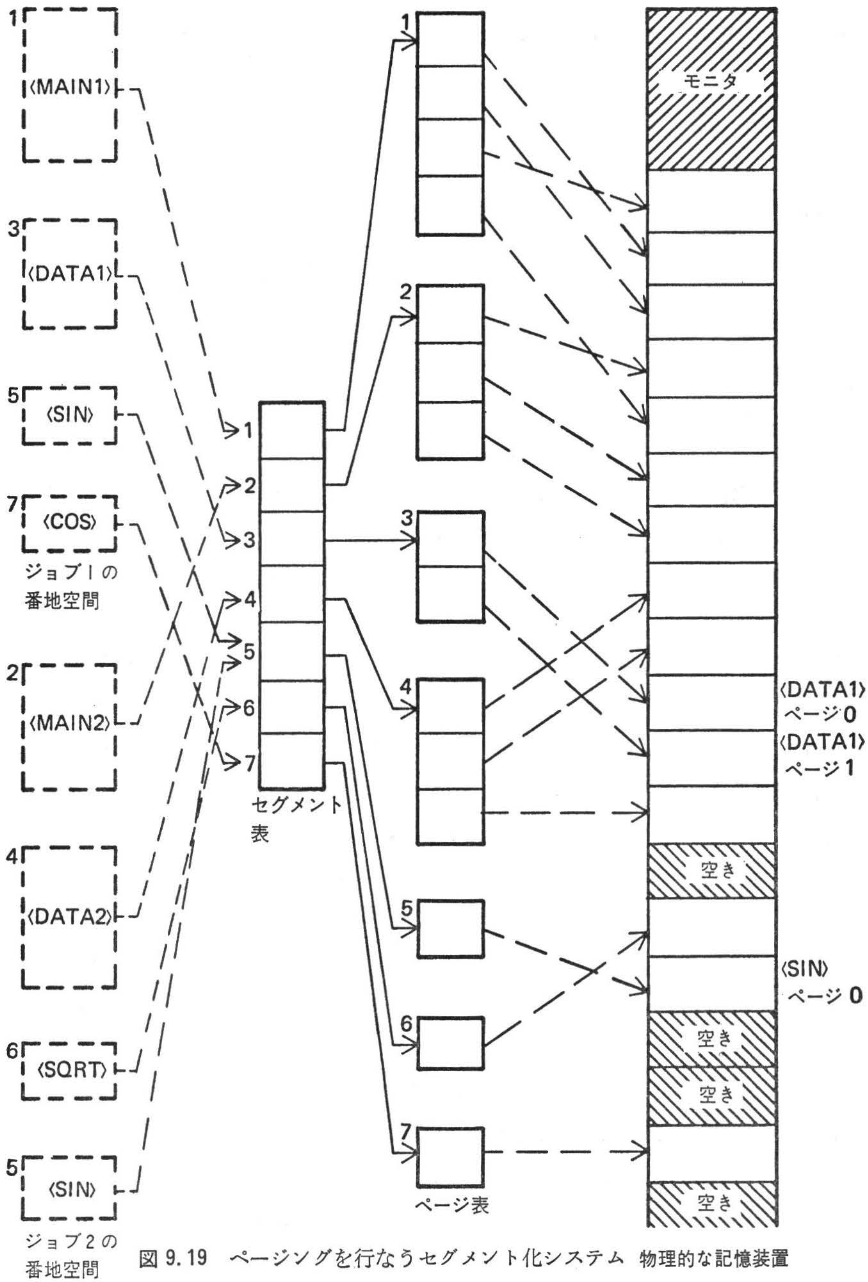


図 9.19 ページングを行なうセグメント化システム 物理的な記憶装置

ーティング・システムにはかなり複雑な記憶管理が必要である.

第 3 部

9.3 プロセッサ管理

前節までに述べた多重プロセッサ構成と記憶管理とを駆使して仕事量の増大の要求を達成するには多重プログラミングを行なわねばならない．本節では多重プログラミングと多重処理に関連する機構を述べる．この議論は広い範囲の計算機システムに適用できるものである.

サブルーチンを共有したり，割込みを含む非同期動作を制御することが必要とされると，プロセッサ管理には複雑な問題が生じる．これらの機構を論じるのには次の言葉を用いると都合がよい.

プロセスとは基本的には実行中のプログラムをいう．簡単な場合には１つのプロセスは１人の利用者のジョブに対応する．番地空間と名前空間を同義語として用いているが,これはプロセスによって作られる番地から成っている.記憶空間または物理的な記憶装置は記憶場所に対応した番地の組から成っている.

プロセッサとは一連の命令を実行することのできるハードウェアの装置をいう．プロセスの他の見方は，ジョブを実行している間にプロセッサが番地空間を通った“経路”であるとするものである．このように，種々の場合にプロセスはオペレーティング・システムの中，利用者プログラムの中，または共有されたプログラムの中に入っている．**多重プログラミング**とは１つのプロセッサで２つ以上のプロセスが混り合って実行されることをいう．**多重処理**とは２つ以上のプロセスを多重プロセッサのコンピュータ・システムで同時に実行することをいう.

プロセッサ管理の基本的な問題は，オーバヘッドが最小になるようにプロセッサをプロセスに割付けることである．プロセスは，実行中，閉塞中，実行待ちの３つの内のいずれかの状態をとる．プロセッサによって一連の命令が実行されているときプロセスは**実行中**であるという．実行を続けるためにある事象の発生を待っているときプロセスは**閉塞中**であるという．実行状態になるため

のすべての条件が満たされプロセッサが割付けられるのを待っているときプロセスは**実行待ち**であるという．プロセスが閉塞中または実行待ちのときは実行が中断されている．プロセスの“状態図”は**図 9.20** に示されている．

一般に，プロセスよりもプロセッサの個数の方が少ない（たとえば，通常，プロセッサは1台だけである)．したがって，プロセッサを配給してやらなければならない．実行待ち状態のプロセスの一覧表を**実行待ち表**という．実行待ちのプロセスはもし十分な台数のプロセッサが存在すればすべて実行される．

実行中から閉塞中への遷移は，通常，外部条件によって決定され(たとえば，I/O チャネルや I/O 装置の速度)，これらの遷移に対しては大部分オペレーティング・システムの制御がおよばない．したがってプロセッサ管理の問題は次のようにまとめることができる：

1. 各プロセスの状態を管理する．
2. 実行のために実行待ち表からプロセスを選ぶ．
3. 割当て時間を使い果たしたプロセスの実行を中止する．
4. プロセス間の通信を仲介する．

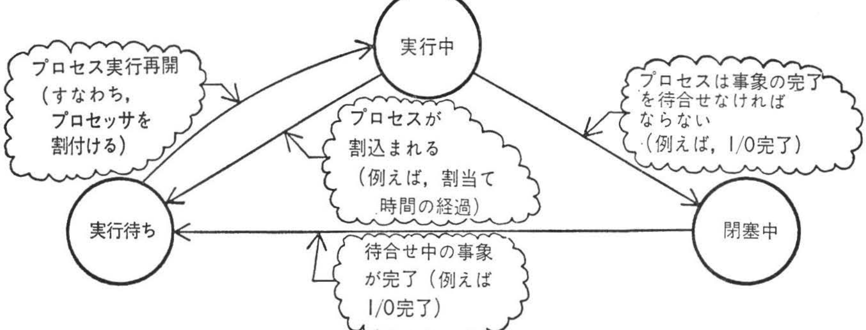


図 9.20　プロセスの状態遷移

これらの仕事を行なうオペレーティング・システムの構成要素は Saltzer によって**トラフィック制御**と呼ばれている(第10章参考文献)．実行のために実際にプロセスを選択する部分を**スケジューラ**と呼び，トラフィック制御から切離して考える方が都合がよい．トラフィック制御は (1)プロセスの状態を変化させ

る機構と，(2)最小のオーバヘッドですべての利用者に公平にプロセッサを割付けるスケジューラとを持つ.

9.3.1 スケジューラ

スケジューラは，プロセッサ時間を要求している各種の実行待ちプロセスに対して乏しいプロセッサ時間を配給するために必要とされる．スケジューラは次の２つの機能を果たす：

1. どの実行待ちプロセスを次に実行するか選び，
2. **タイム・スライス**（一回に割当てられる最大プロセッサ時間）を決定する.

プロセスが割当てられたタイム・スライスを使い果たすと時計割込みが起こり，**図9.20**に示したようにプロセスは一時的に実行待ち状態に戻る.

一般に，１つの実行中のプロセスが中断され，閉塞または実行待ち状態となると，実行待ち表から１つのプロセスが選ばれて実行状態とされ，空いたプロセッサを割当てられて使用する．スケジューリングを行なうには2とおりの方法がある．これらは共に実行待ち状態のプロセス一覧表（**実行待ち表**）を用いる．１つの方法はプロセスが実行待ち表に入れられる度にこの表をスケジューラが並べておく方法である．この方法ではプロセッサが空くとスケジューラは表の最初のプロセスを取り出し，実行状態にするだけでよい．他の方法はプロセスを実行待ち表の中にランダムに入れておくものである．プロセッサが空くと，スケジューラは表全体を走査して，どれを実行するかを決定する.

スケジューリングの方法やシステムの目的によって，適当なスケジューリングの機構を選べばよい.

実行待ち表に2つ以上のプロセスが入っているときには，スケジューラはスケジューリングの方針にしたがって，どのプロセスを実行するかを決定しなければならない．これはしばしばプロセスに異なった優先順位を割当てることによって行なわれる.

この方針と優先順位は次のような要素によって定められる：

1. **料金**　高い料金を支払うことによって高い優先順位を“買う”．高い優先順位は高速の処理を意味し，よい席を求めるために給仕にチップを払

うことの工学版である．

2. **必要度** 原子炉の爆発を緊急に防止するとか，遅れた計画を促進させたりするためのジョブなど，利用者は重要度に応じて優先順位が割当てられる．

3. **価値** ある場合にはシステム自身で優先順位を割当てるのが都合のよいことがある．たとえば，短いジョブには高い優先順位を自動的に割当てることができる．他の例は，1つのジョブが入力／出力を多量に要求し（**I/O 制限**），1つのジョブはほとんど入力／出力を必要としない（**CPU 制限**）ような場合，これらを同時に実行して平衡のとれた負荷とするために優先順位を指定することである．すべてのジョブが極端に片寄ると，I/O チャネルが過負荷となり CPU がアイドルになったり，またはその逆になったりする．したがって，システムで最大の仕事量を得るためには，適当にジョブを混合し平衡のとれた負荷となるように優先順位を割当てるのが好ましい．同様にシステムは，資源（先に割当てられたテープ駆動装置など）を専有しているジョブを早く完結させ，その資源を他のジョブが使用できるよう解放するために，そのジョブの優先順位を大きくしたいことがある．

多くの配給制度の場合と同じように，何が“公平”で，“効率がよく”，“経済的”な方法であるかについては大いに議論のあるところである．読者はスケジューリングの方針についての詳細を参考文献（第10章）で調べてみてほしい．幾多の計算機“民俗学”がこの分野で発展している．

スケジューリングの方針を状態遷移図で示すと大変都合がよい．**図 9.20** よりもっとこみいったスケジューリング・アルゴリズムが**図 9.21** に示されている．この図では**実行待ち**と**閉塞中**の状態をさらに分割してプロセッサの割当てを行なっている．

このスケジューリング・アルゴリズムは，ページングによる記憶管理を行なうタイム・シェアリング・システムで使用するのに**適した**方式を簡単化したものである．特に，閉塞中のプロセスは3つに分割され（端末 I/O 閉塞中，ディスクまたはテープ I/O 閉塞中，ページング I/O 閉塞中），実行待ちプロセスは3つに分割されている（CPU 制限，高順位 I/O 制限，中順位 I/O 制限）．スケ

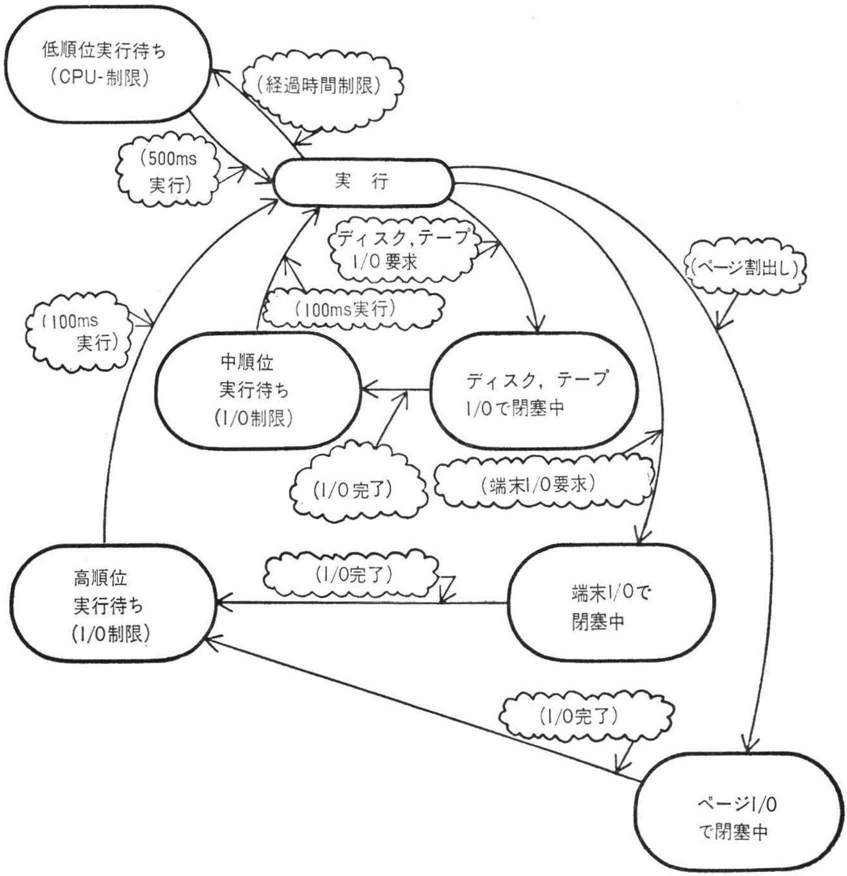


図 9.21　プロセスの状態遷移のスケジューリング

ジューリングの方針は高順位の実行待ち表から1つのプロセスを選択することである．該当するものがなければ，プロセスは中順位の実行待ち表から選ばれる．低順位のプロセスはそれより高い順位のプロセスが存在しないときのみ実行される．

このアルゴリズムには多くの方針が組込まれている．たとえば，I/O を要求せずに 100 ミリ秒（約 100,000 命令）以上実行すると，一時的に CPU 制限のジョブであるとみなされる．I/O チャネルや I/O 装置を動作中にしておくため

に，I/O 制限プロセスはつねに CPU 制限のプロセスよりも先に実行される．I/O 制限プロセスが閉塞されると，次に CPU 制限のプロセスがプロセッサを与えられ，I/O 要求を出して閉塞されなければ，停止されるまで 5 倍の時間（500 ミリ秒）実行を許される．読者は**図 9.21** を調べてこのアルゴリズムに含まれている他の方針を識別してみよ．またこれらの方針の欠点をも調べよ．

9.3.2 トラフィック制御

トラフィック制御の仕事は，各プロセスの状態を管理し，プロセスを中断したり再開したり，プロセス間通信を仲介したりすることである．

中断されたプロセスを再開するためには，トラフィック制御が，閉塞される時点で**状態語**と呼ばれるプロセスの状態の“スナップ”を保存しておかなければならない．プロセスの状態語はレジスタの内容，命令の番地，条件コードなどを含む．トラフィック制御は，たとえば I/O 要求などによって陽に呼出されることもあるし，I/O 完了割込み，時計割込み，またはページなし割込みなどの割込みによって陰に呼出されることもある．プロセスは，もっと記憶域を要求したり，テープ駆動装置を要求したりした場合などによく起こるが，これらがすでに使用中で，現在直ちに使用できない資源をオペレーティング・システムに要求したときにも閉塞状態となる．後でその資源が使用可能となると，トラフィック制御は前に閉塞されたプロセスを実行待ち表に入れる．すべての状態遷移はトラフィック制御で処理されるということが基本的なことである．このようにしなければ混乱が生じたりプロセスが“なくなってしまったり”する恐れがある．

動作中のプロセス相互間で，プログラムの制御のもとに通信を行ないたいことがある．すなわち，プロセスは他のプロセスを実行待ち状態または閉塞状態にしたいことがある．これが認められる場合には，トラフィック制御は呼出しを受理し，他のプロセスを閉塞状態にすることができなければならない．同様にトラフィック制御は，“条件が満たされ，プロセスは実行を再開できるようになったよ”などと閉塞中のプロセスに対して通信することを望むプロセスからの呼出しをも受理してその通信を仲介しなければならない．この機能は，通常，トラフィック制御によって行なわれ，**プロセス間通信**と呼ばれている．

プロセッサの割付けとプロセス間通信に関しては，プロセッサとプロセスとの同期，および競争と“すくみ”の2つの問題点が存在する．これらは次節で詳しく述べられる．

9.3.3 競争条件

競争条件は．2つのプロセスのスケジューリングが非常に微妙でスケジューリングの順番によっては計算結果が異なるような場合に起こる．競走条件は，2つ以上のプロセスの間で陽にまたは陰にデータや資源を共有するときに発生する．**図9.22**は簡単な形の競争を示している．

図9.22では2つのプロセスが(多重プログラミングで)実行中であると仮定する．各プロセスは1台の印刷機で行を印刷する要求を出すものとする．プロセス1および2のスケジューリングによってプロセス1の印刷出力とプロセス2の出力は別々に印刷されるのが普通である．しかし，スケジューリングの方法によっては両方の印刷結果が紙の上で混り合うかも知れない．

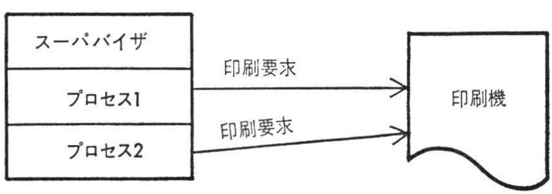


図9.22　単純な競争条件

この問題を解決する1つの方法は，プロセスが共有資源（この場合印刷機）を使用する前に，その使用を陽に**要求**することである．資源の使用がすべて終了していれば(たとえば，すべての印刷が完了したとき)，この資源はプロセスによって**解放**される．**要求**と**解放**の動作は，通常，トラフィック制御で処理されるオペレーティング・システムの機能の一部である．プロセスがすでに使用中の資源を“要求”すれば，このプロセスは自動的に閉塞状態となる．後でその資源が“解放”されて使用可能になると，そのプロセスは資源を割当てられて実行待ち状態となる．

プロセスの実行を調整するこの技法はしばしば**インターロック**とか**同期**と

か呼ばれる.

インターロックを効果的に使用するには協同動作が重要である. もしプロセスが割付けられた資源を解放することを“忘れた”ならば, その資源はそれ以後二度と他のプロセスで使用することができなくなる. そこで実際には, 1つのプロセスが終了すると, オペレーティング・システムはそのプロセスに割付けられていたすべての資源を解放する. 前述の印刷機の問題に対する別の解決法は装置管理の節で論じる.

9.3.4 手 詰 ま り

手詰まりとは2つ以上のプロセスが, 使用できないことを知らずに, その資源を待っている状態をいう.

要求と解放の動作はすべての競争条件を完全には除去しない. 事実, 時には**手詰まり**とか**すくみ**として知られる致命的な競争を起こすことがある. **図 9.23** の2つのプロセスを考えてみよう.

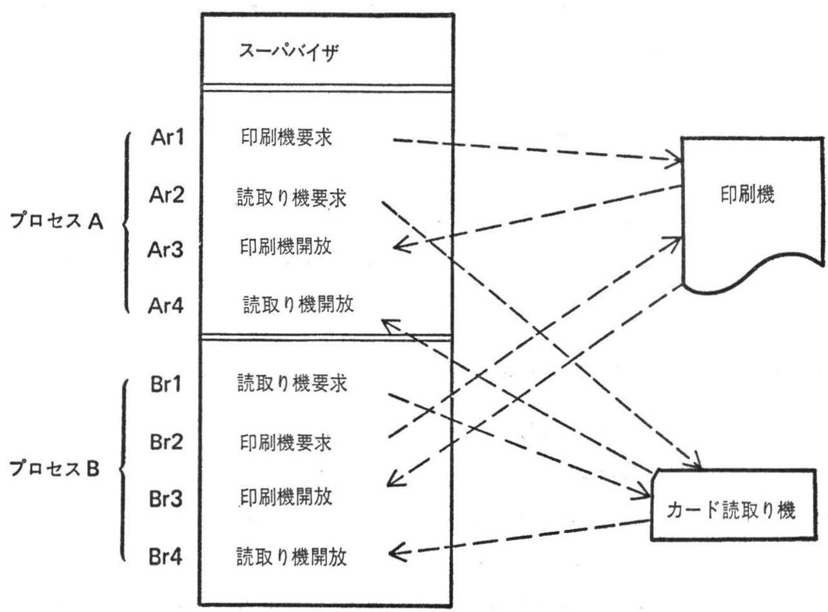


図 9.23 手 詰 ま り の 状 態

プロセスAとBは要求と解放の動作によって印刷機とカード読取り機を共有している．プロセスのスケジューリングは独立であるから，要求と解放の動作は次のように種々の異なった順番に混ざって起こることがある．

(1) Ar1, Ar2, Ar3, Ar4, Br1, Br2, Br3, Br4

(2) Br1, Br2, Br3, Br4, Ar1, Ar2, Ar3, Ar4

(3) Ar1, Ar2, Br1, Ar3, Ar4, Br2, Br3, Br4

注目してほしいのはこれらの順番は実際に起こり得るということである．(3)ではプログラムAが Ar4 を出すまではプログラムBが一時的に Br1 で閉塞されている．

さて，Ar1（プロセスAが印刷機を要求）と，Br1（プロセスBが読取り機を要求）が出されたとする．次に何が起こるであろうか．もし Ar2（プロセスAが読取り機を要求）が起こると，読取り機はプロセスBですでに使用中であるから，Aは閉塞されなければならない．次に Br2（プロセスBが印刷機を要求）が起こると，印刷機はプロセスAで使用中であるから，Bもまた閉塞されなければならない．ここで手詰まりになってしまう．プロセスAもBも閉塞され，各々は必要な資源を相手が解放するのを待っている．

手詰まりの問題は多重プログラミング・システムにおいて種々の場合に起こり得る．不幸にして現在のところこの問題を解決する単純でしかも効率のよい技法がない．最も単純な技法は，多分必要とされるであろう資源を，ずっと後の処理にならないと必要にならなくても，同時に全部要求することである．しかし，明らかにこの方法は効率のよいものではない．この分野では多くの研究がなされている．

9.3.5 多重プロセッサ・システム

仕事量を増大させるために，多くの計算機システムにおいてはプロセッサが2台以上用いられている．初期の多重プロセッサ・システムでは2台目のプロセッサは，たとえば I/O チャネルなどの，特定の機能を持ったものであった．多重処理システムは1つの大規模な CPU と幾台かの周辺プロセッサ（CDC 6600）の考えに発展した．周辺プロセッサは非常に複雑な仕事，たとえばディスプレイを動作させることなどを行なっている．多重処理の3番目の型は HIS 645 に

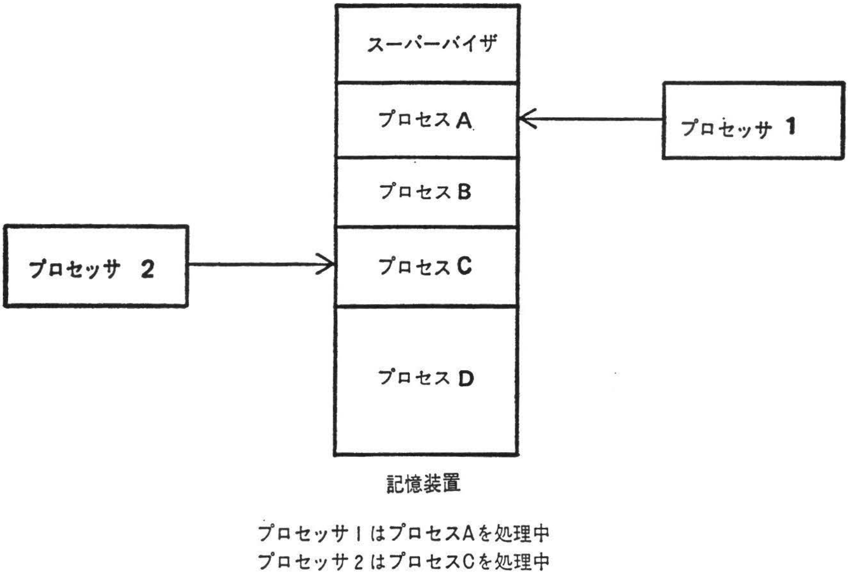
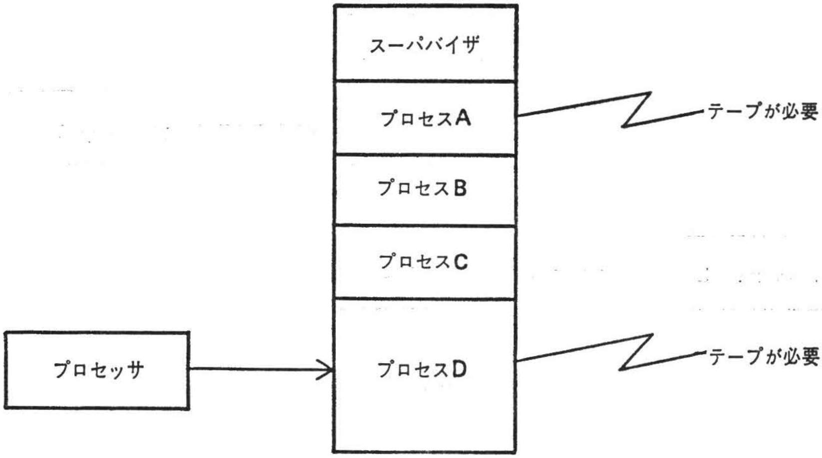


図 9.24 多重プロセッサによる多重プログラミング

見られる．このシステムには2台以上の等しい能力の CPU が設置できる．4番目のより一般的な型は計算機ネットワークであり，このネットワークには多数の種類の異なる計算機が，しばしば遠距離から，接続される．

図 9.24 は多重プログラミングに対して複数のプロセッサが使用される様子を示したものである．通常幾つかのプロセスが記憶装置に入っている．現実にはプロセスはしばしばページに分けられていて，一時にはその一部分だけが記憶装置に入っている．このようにするとシステムの中でアクティブなプロセスの数を非常に大きくすることができる．1台のプロセッサが1つのタスクに割当てられ，それが閉塞するまで実行する．タスクが閉塞されると，プロセッサは他のタスクを選択し，処理を続行する．後で閉塞を起こした条件が満たされると，1台のプロセッサがいつかはそのプロセスに割当てられる．このとき前と同じプロセッサである必要はない．

プロセッサが複数台のシステムでもプロセッサが1台のシステムでもプロセスの閉塞は起こり得る．**図 9.25** ではプロセッサが1台の多重プログラミング・システムを示す．プロセスAとDはテープ駆動装置の使用を要求している．



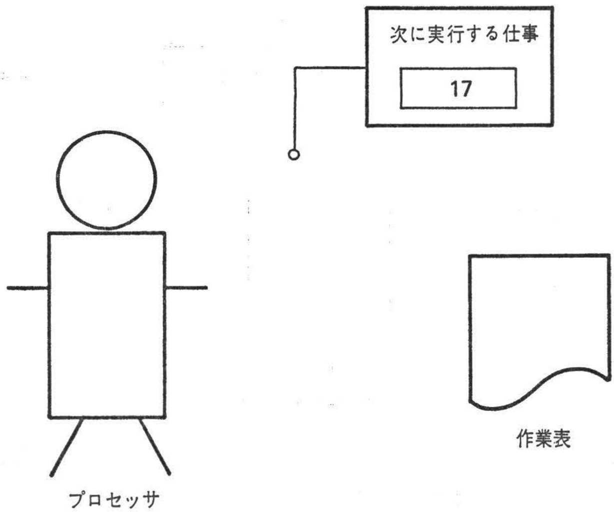
ただ1つのテープ装置があるシステムで2つのプロセスAとDが共にテープ装置を必要とすると，1つのプロセス(たとえばA)にこれが割当てられ，他のプロセス（D）は閉塞される．処理はプロセスA，BおよびCに関して進められる．プロセスAが完了すると，テープ装置は他に割当てることができるようになり，再びDが実行される．プロセッサは停止することなく単に‘論理的な状態’が変化する．

図 9.25　単一プロセッサによるインターロック

テープ駆動装置はただ1台しか使用できないとし，プロセスAが先にそれを要求したとする．装置はプロセスAに割当てられる．後に，プロセスDがテープ駆動装置を要求すると，Aがテープ駆動装置を解放するまでDは閉塞状態とならなければならない．

閉塞状態はプロセスの属性であり，プロセッサの状態ではない．プロセッサはプロセスA，BおよびCの間で切換えられ，プロセスDを無視する．通常，プロセスDは記憶装置の一部を占有したままになっているが，システムによってはその部分が主記憶装置から2次記憶装置へ“ロール・アウト”または“ページ・アウト”される．この場合には，プロセスDが閉塞状態になることは，資源の再分配に論理的な影響を与えるだけであり，資源の無駄使いとはならない．

一般に，多重プロセッサ・システムはプロセッサが1台のシステムと同様にジョブを実行する．プロセッサの処理を調整するために2，3の特別な技法が



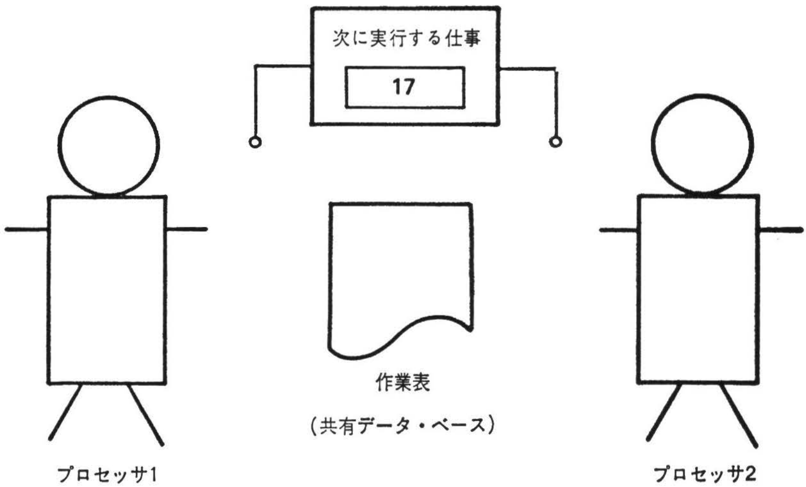
実行しているプロセスが完了したり中断されると；

1. 作業表の中でそのプロセスを探し，
2. “完了”または“中断”と記入し，
3. ‘次に実行する’番号が17であることを知り，
4. 作業表の中でプロセス17を探し，
5. そのプロセスが“実行中”であると記し，
6. ‘次に実行する’番号を18とする．

図 9.26 単一プロセッサのスケジューリング

使用される．この技法の1つは**ソフトウェア・プロセッサ・ロックアウト**である．これは複数のプロセッサによるデータ・ベースの使用を調整するものである．ここではこの技法を，まず，1台のプロセッサでのソフトウェア上の問題点を示し（**図 9.26**），次に2プロセッサ・システムの場合を示し（**図 9.27**），次に解決法を示して（**図 9.28**）論じて行こう．

システムにはすべてのプロセスとその状態とを一覧表にしている重要なデータ・ベースが存在する．この表はジョブ行列，タスク行列，実行待ち表，または作業表と呼ばれている．プロセスが閉塞されると実行待ち表の欄を見つけてそこにプロセスの状態を格納する．そして次に実行すべきプロセスが決定される．この例では，単純なラウンド・ロビンの選択法を仮定している．選ばれた



2つのプロセッサが同時にプロセスを完了したとする：

1. プロセッサ1は作業表の中で実行していたプロセスを探す.
2. プロセッサ2は作業表の中で実行していたプロセスを探す.
3. プロセッサ1はそのプロセスに“完了”と記入する
4. プロセッサ2はそのプロセスに“完了”と記入する
5. プロセッサ1は‘次に実行すべき’番号が17であることを知る.
6. プロセッサ2は‘次に実行すべき’番号が17であることを知る
7. プロセッサ1はプロセス17を作業表の中で探し“実行中”であると記す.
8. プロセッサ2はプロセス17を作業表の中で探し“実行中”であると記す.
9. プロセッサ1は‘次に実行すべき’番号を進め18とする.
10. プロセッサ2は‘次に実行すべき’番号を進め19とする.

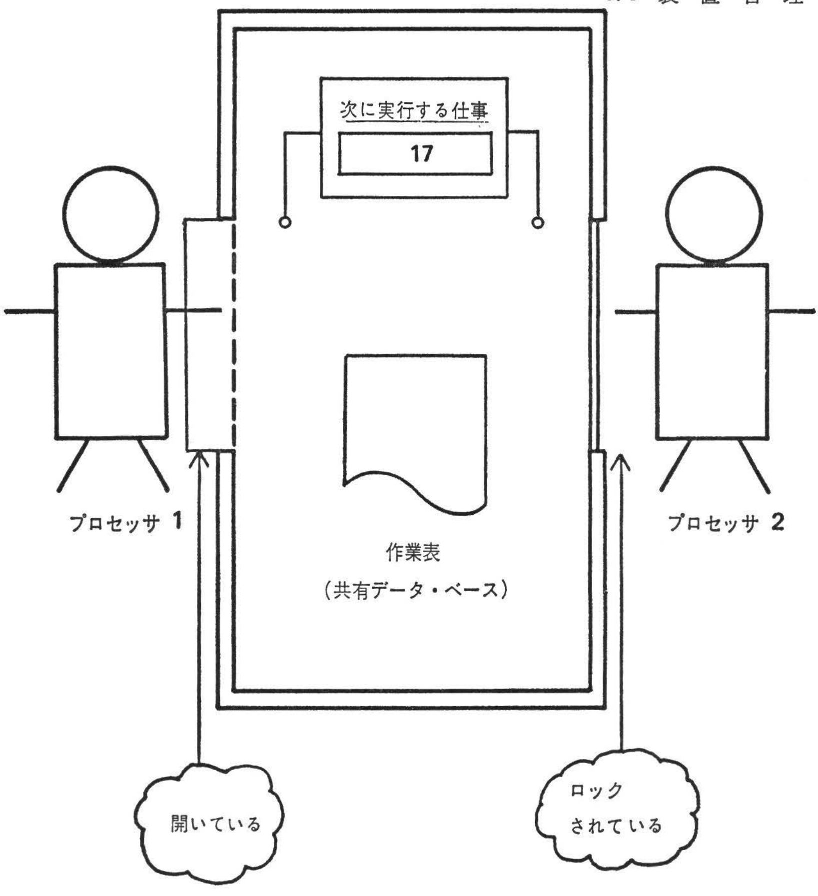
ここで2つのプロセッサは同じプロセスを同時に実行することとなり，プロセス18は飛ばされてしまう.

図 9.27　多重プロセッサのスケジューリング

プロセスは検査され，状態をレジスタなどに設定して実行される．またその後のプロセスが次に実行されるという表示を行なっておく.

上述の方式は1台のプロセッサの場合にはうまく働くが，複数のプロセッサの場合は**図 9.27** に示したような問題が生じる.

別々のプロセッサで実行されている2つのプロセスがほぼ同時に閉塞されたとする．プロセスの状態を保存した後に各プロセッサは次に実行すべきプロセスを決定する．両方のプロセッサは独立に同じアルゴリズムを使用しているか



一時に1つのドアのみが開く
図 9.28 ロックアウトを行なう多重プロセッサのスケジューリング

ら，ここで同じプロセスを選び出すことになる．さらにまた，次に実行すべきプロセスの指定を調整するに当って，たとえば**図 9.27** に示したようにプロセスを１つ抜かして作業表を狂わせてしまうことも起こり得る．

この問題は，複数のプロセッサが共通のデータ・ベースを非同期に処理をするときに起こるものである．通常，オペレーティング・システムには，記憶装置割付けマップ，ページ割付けマップ，I/O 一覧表など，多数のデータ・ベースがある．

ふだんは独立に動作しているプロセッサでも，共通のデータ・ベースを参照するときには，同期を取らなければならない．**図9.28**はこの論理的な処理を示している．

データ・ベースを参照する前に，プロセッサは特別の“ロック・ビット”を検査する．もし *on* でなければ，そのデータは使用中でないから，プロセッサはビットを *on* としてこのデータ・ベースに対して処理を開始する．処理を終了したとき，プロセッサはビットを *off* にする．第2のプロセッサが途中でこのデータ・ベースの参照を要求したときは，“ロック・ビット”が *on* であることを知り，“ロック”が解かれるまで待たなければならない．この条件のとき，第2のプロセッサは**ソフトウェア・プロセッサ・ロックアウト**の状態にあるといい，一時的にアイドルになる．第2，第3，第4と何台でもプロセッサが同時にこの状態になることも起こり得る．

閉塞はプロセスの現象であり，スーパバイザは資源を再割当てし，プロセッサを停止させずに他のプロセスに切換えることができた．ソフトウェア・プロセッサ・ロックアウトはスーパバイザの中で起こる．他のプロセッサとの間で矛盾を生じる危険を犯さないためには，プロセッサを停止させること以外にはない．このようにして，物理的な資源であるプロセッサはソフトウェア・ロックアウトによってアイドルにされることがある．

ソフトウェア・プロセッサ・ロックアウトが多重プロセッサ・システムの仕事量にどのような影響を与えるかを解析することはきわめて複雑な問題である．このようなシステムの複雑な動的特性は，実際に測定してみて初めて正確にその影響が測定できる．しかし，この状態を近似できる確率的な解析用モデルを作ることも可能である．プロセッサ管理の詳細な組込みは本書の範囲を越えるので，これ以上の議論は第10章の参考文献を参照してほしい．

第 4 部

9.4 装置管理

現代の計算機システムは，印刷機，カード読取り機，磁気テープ，磁気ディ

スクおよびドラムなどの**周辺装置**と呼ばれる多種多様の“入力／出力”装置を持っている．大規模な設備では経費の半分が周辺装置に当てられる．したがって，これらの装置を最も効率よく使用することが望まれる．

9.4.1 装置の特性

およそ考え得るものは計算機の周辺装置となり得るし，また周辺装置とされてきた．製鋼所からレーザ光線に至るまで，レーダから光学機械に至るまで，また温度計から宇宙船に至るまできわめて広範囲にわたっている．しかし，計算機設備の多くはこれらのうちのごく一部の周辺装置のみを利用している．

周辺装置は一般に次の２種に分類できる：

1. 入力または出力装置
2. 記憶装置

入力装置によって計算機は外界を“知覚”したり“さわって”みたりする．これらは温度計やレーダなどのこともあるが，普通多いのは，穿孔されたカードや，穿孔された紙テープや，タイプライタ端末のような打鍵されるメッセージを読込む装置である．**出力装置**によって計算機は外界に“影響”を与えたり“制御”を行なったりする．これらは温度制御つまみやレーダの方向制御などのこともあるが，普通多いのはカードに穿孔したり，紙テープに穿孔したり，紙の上に文字や数字を印刷したり，タイプライタ端末などのように印字したりする装置である．これらの入力および出力装置は実際のI/Oプログラムの例と共に第9.1節でかなり詳細に論じられている．

記憶装置によって計算機は情報を格納することができる．格納することを一般に**書込み**といい，後でこの情報を取り出す（**読出し**）ことができる．概念的には記憶装置は人間が使用している紙片と鉛筆と消しゴムなどの記録の手段と類似している．

記憶装置には順アクセス型と直接アクセス型の２つの型がある．

順アクセス型記憶装置は物理的に情報を順番にアクセスすることによって特徴づけられる．任意の項目へのアクセスは“逐次探索”法で行なうことが必要であり，アクセスのために，平均的には，格納されたすべての情報の半分を読出す時間がかかる．磁気テープは順アクセス型記憶装置の最も一般的な例であ

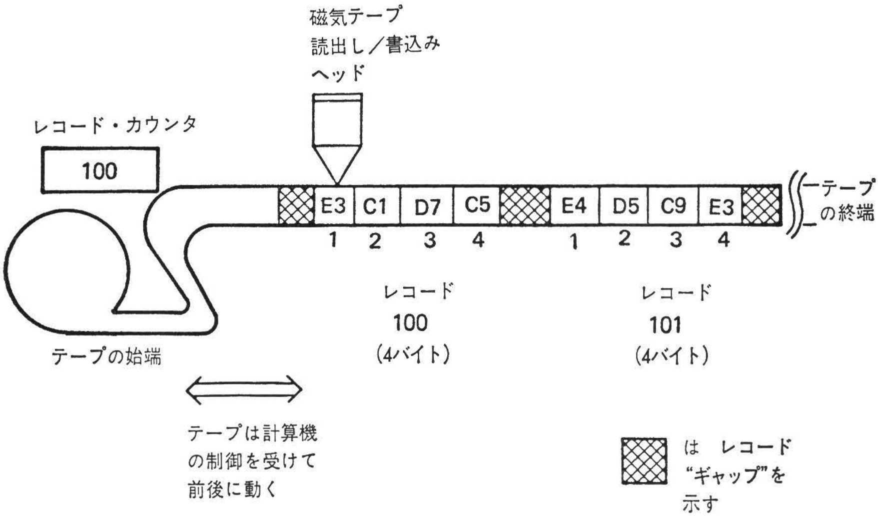


図 9.29 順アクセス型記憶装置

る．磁気テープはオーディオのテープ・デッキやカセット・テープと同じ原理によってできているが，音楽や音声を録音する代りに2進情報，一般にはバイト，が格納される．**図 9.29** に典型的な順アクセス装置が示されている．

情報は1バイトずつではなく，通常**レコード**と呼ばれるバイトのかたまりとして格納される．一般に，レコードの長さは1バイトから 32,768 バイトというように任意である．**図 9.29** ではすべてのレコードの長さは4バイトである．レコードは標準のオーディオ・テープ・デッキに“録音された単語”と類似していると考えることができる．各レコードは，第1レコードは1，第2レコードは2というように，テープ上の物理的な位置で識別することができる．**図 9.29** のレコード・カウンタは，オーディオ・テープ・デッキの“時間計”または“フィート数”と同じ機能を果たす．テープが最初の位置に設定されていて（すなわち，完全に巻戻されている)，レコード番号 101 を読出したいとすると，レコード番号 101 に達するためには，それまでのレコード（1, 2, …, 99, 100）をすべて読み飛ばさなければならない．最近の磁気テープは，1インチに最大 1,600 バイト記録することができ，1秒間に最高 200 インチの速度で移動することができる．しかしこの速度でも，標準の 2,400 フィートのテープを

最初から読み始めて中央のレコードに達するには約1分かかる．順アクセス記憶装置は普通，アセンブラ（第3章）やローダ（第5章）のパス1で作られる中間ファイルなど，順アクセスのみを必要とするような応用に使用される．

直接アクセス型，または**乱アクセス型**記憶装置（DASD）は，物理的に順アクセスを必要としないことで特徴づけられる．計算機システムの主記憶装置は直接アクセス型記憶装置の例である．記憶装置の任意のバイトは，他のバイトを“読み飛ばす”のを待つことなく直接参照することができる．一般に，最近の直接アクセス型記憶装置は完全な直接アドレス（たとえば，主記憶装置）と順アクセス（たとえば，磁気テープ）とを折衷した物である．**図 9.30** は磁気ドラムと同様の直接アクセス型記憶装置を示している．

磁気ドラムは，おおざっぱにいえば円筒に磁気テープの帯を隣り合わせに幾つも巻きつけて，各帯の始めと終りを接続したものと考えることができる．各帯は**トラック**と呼ばれ，別々の読出し／書込みヘッドを持っている．ドラムはつねに高速で回転しており，レコードは繰り返し読出し／書込みヘッドの下を

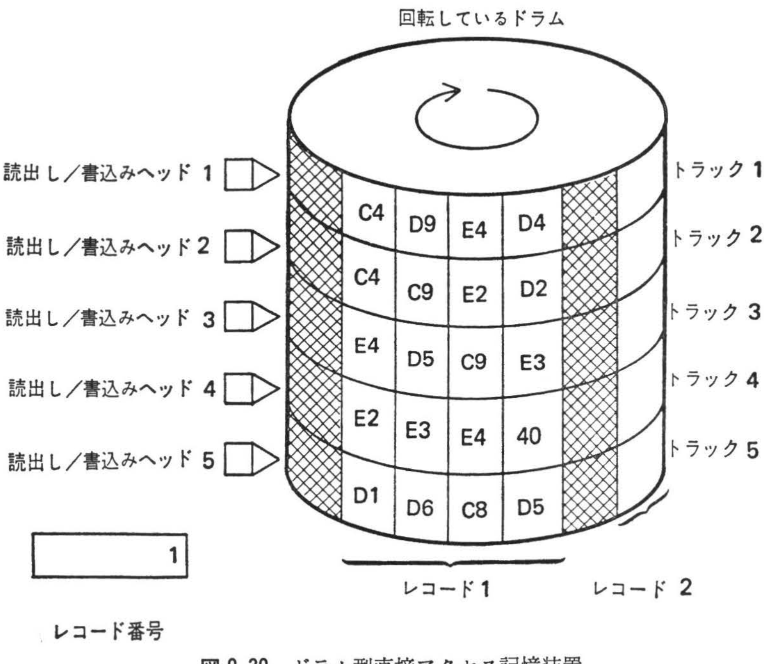


図 9.30 ドラム型直接アクセス記憶装置

通過する（たとえば，レコード1，レコード2，……，再び，レコード1，レコード2，…というように）．各レコードはトラック番号とレコード番号とで識別される．たとえば，**図 9.30** においては，レコード（2, 1）は X′C4C9E2D2′ であり，レコード（5, 1）は X′D1D6C8D5′ である．磁気ドラムは高速で回転し，数百もの読出し／書込みヘッドを持っている．したがって，磁気テープ1巻のレコードを乱アクセスするとき1分かかるのに比べて，乱アクセスの読出しや書込みは5～10ミリ秒で実行することができる．

直接アクセス記憶装置には多くの種類がある．磁気ディスクは磁気ドラムにきわめて類似しているが，一連の同心円のトラックを持つ円板を1つの読出し

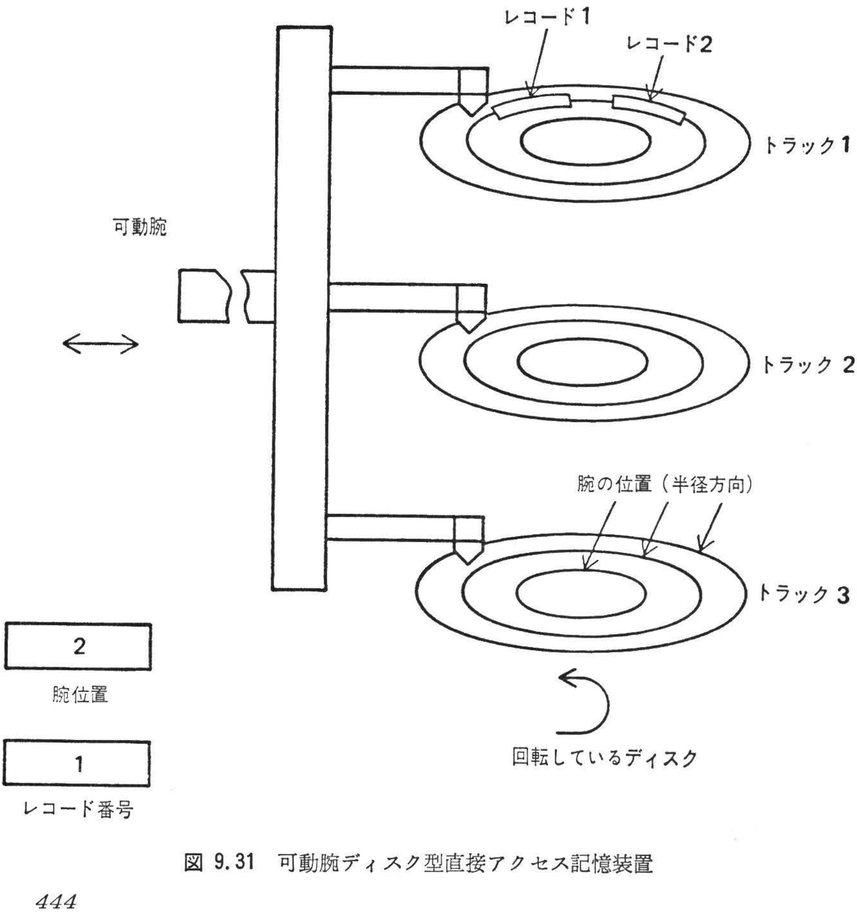


図 9.31 可動腕ディスク型直接アクセス記憶装置

／書込みヘッドに一枚ずつ使用している．これは蓄音器のディスクと類似している．読出し／書込みヘッドは高価であるので，装置によってはトラックごとに専用のヘッドを持たず，ヘッドが物理的にトラックからトラックへ移動する．このような装置を移動腕または移動ヘッド直接アクセス型記憶装置と呼ぶ．

図 9.31 に示された移動腕の直接アクセス型記憶装置に記憶されているレコードを識別するためには，腕の位置と，トラック番号と，レコード番号とを指定しなければならない．腕の位置は直径方向の変位であり，レコード番号は円周方向の変位である．移動腕ディスク型の直接アクセス型記憶装置は，10〜50 トラックと 200〜400 の腕の位置を持ち，100,000,000 バイト以上のデータを格納することができる．

9.4.2 装置管理の技法

単純なバッチ・モニタ・システムでの装置割付けはきわめて能率の悪いものである．このシステムでは一時に１つのジョブだけが実行され，通常，ジョブで使用されない大半の装置は遊んでいる．多重プログラミング・システムは，実行中のすべてのジョブによって必要とされる装置が利用される．このようなシステムの目的は，すべての周辺装置の利用率を最大とし，仕事量を多くすることである．同時に実行されるジョブは周辺装置の‘溜り場’から資源を割付けられる．この割付けには，専用と共用との２つの形態がある．

専　　用

カード読取り機，磁気テープ，および印刷機などの装置はその性格上，一時に１つ，順番にしか使用することができない．たとえば，３つのジョブの印刷出力を紙の上に混合して出力しても一般には何の役にも立たない．これらの装置は一時に１つのジョブで専用されなければならない．

直接的な割付け方法は，このような装置をジョブに**専用**させ，その装置が使用できるときだけ実行を開始することである．このようにするとは多重プログラミングと優先順位のあるスケジューリングが複雑になる．明らかにあい路となる部分が存在する．たとえば，ほとんどすべてのジョブは何らかの印刷出力

を行なうので印刷機を使用しなければならない．しかしシステムに2台しか印刷機がないとしたら，単純な専用装置割付けを行なうと多重プログラミングを2ジョブ以下に制限してしまうことになり，きわめて能率が悪い．このあい路を解消する一般的な方法が**スプーリング**（SPOOLING：オンライン同時周辺動作）であり，印刷を行なおうとするすべてのジョブ出力は，記憶装置や直接アクセス装置（たとえば，ドラム）に，ジョブが終了するまで保存される．ジョブ実行の間に間欠的に印刷することをせずに，ジョブが終了すると保存されていた出力は高速で印刷される．すべてのジョブの印刷出力要求の合計が印刷機の能力を越えない限りは，保存された出力を印刷できる．類似の方法がカード読取り機の入力についても使用できる．すなわち，ジョブのカード入力はオペレーティング・システムによって高速で読込まれ，ジョブが開始される前に直接アクセス装置に保存される．

共　用

直接アクセス型記憶装置（たとえば，磁気ドラム，ディスクおよびデータ・セル）などの周辺装置は2つ以上のジョブで**同時**に使用することができる．直接アクセス型記憶装置に格納されている情報は直接アクセスでき，複数のジョブが装置の別の場所をアクセスするとき相互にほとんど干渉がない．

共用装置の使用に関しては，オペレーティング・システムで処理しなければならない2つの問題点がある．

第1：装置を複数のジョブで共用することによって起こる相互干渉はほとんどないが，一時には1個所のアクセスしかできない（すなわち，同時には，1台の装置について1個所しか読出したり書込んだりできない）．装置によっては，1回の参照は数ミリ秒から数秒かかる．したがって，同時に2つ以上のジョブが同じ装置を参照しようとすると，オペレーティング・システムは交通整理の役を果たさなければならない．事実，この機能はしばしば**I/O トラフィック制御**と呼ばれる．オペレーティング・システムのI/O トラフィック制御は大きなシステムでは通常かなりの容量を占めるが，その効率は多重プログラミングの効率と仕事量に多大の影響を与える．またI/O チャネル（I/O 計算機）は

容量に制限があり，頻繁に使用されるならばスケジューリングが必要であるので（たとえば，すべてのテープ，ディスクおよびドラムが同時に動作すれば，情報転送の処理は通常不能である），専用装置でもこのトラフィック制御の問題に関係する．I/O トラフィック制御の詳細は，装置の特性と CPU の特性に大きく依存するので，本節ではこれ以上論じない．

共用装置に関する第2の問題点は，他の利用者によってあるジョブの領域が故意または悪意によって参照されることを防止する必要があることである．この保護は通常ハードウェアとソフトウェアの機能を組合わせて行なわれる．前節で論じた記憶保護に加えて，多重プログラミングを目的とした計算機は，通常，マスタ／スレーブと呼ばれる（またスーパバイザ／プロブレムとか，特権／非特権とも呼ばれる）2つの動作モードを持っている．**非特権モード**で動作しているときに，入力／出力や保護の設定などのための特権命令を使用すると命令の不正使用であるとみなされ，制御がオペレーティング・システムに移される．オペレーティング・システムは**特権モード**で動作し，計算機のすべての命令を使用することができる．各I/O 要求の正当性は，しばしばI/O ハードウェアの力を借りて，オペレーティング・システムで検査される．

第 5 部

9.5 情 報 管 理

オペレーティング・システムの最も重要な機能は効果的な情報管理である．計算機システムのことをしばしば“情報処理システム”という表現で呼ぶことがある．オペレーティング・システムはそれ自身情報という資源の例である．これはすべてのジョブによって用いられる．情報管理を行なうオペレーティング・システムのモジュールはしばしば**ファイル・システム**と呼ばれる．

ファイル・システムは情報が柔軟にしかも安全に管理されるようにとの目的で設けられたもので，プログラマは，記憶の書式やI/O アクセスなどの物理的な問題と，情報に対する記憶領域の割付けに関する問題から解放される．した

がって，プログラマは情報に関してその論理構造と行なうべき処理だけを考慮すればよい．また，ファイル・システムは利用者間の情報共有を承認されていない参照から情報を保護する機能をも提供する．

ここでは応用のレベルではなくシステム・プログラマのレベルでファイル・システムを取り上げることにする．ファイル・システム，データ管理システム，およびデータ・ベース・システムの区別をはっきりさせておくと今後の議論に都合がよい．

1. ファイル・システムは情報の論理的な編成に関係したものである．ファイル・システムは，オペレーティング・システムのレベルでは“構造化されておらず，意味の解釈されていない情報の集合”を取り扱う．各々の情報の集合のことを**ファイル**という．
2. データ管理システムとは，たとえば，共通指標つき順編成ファイルなどの“情報の編成を行なう”ファイル・システムである．たいていのプログラマはこのレベルの情報管理を行なう．
3. データ・ベース・システムとは，“データ項目の構造と解釈”に関するファイル・システムであり，たとえば“904 便には何人乗客がいるか”ということを質問できる飛行機の座席予約システムなどがある．大半の末端利用者はこのレベルの情報管理を行なう．

ファイルとは，情報単位（**レコード**）を1単位として考えたとき，互に関連した情報単位の集合をいう．レコードとはデータ要素を単位として考えたとき，互に関連したデータ要素の集合をいう．たとえば，在庫管理への応用においては，1枚の送り状の1行は1つのデータ項目であり，1枚の送り状は1つのレコードであり，一揃いのこのようなレコードの組はファイルである．

9.5.1　ファイル・システムの発達

前ファイル・システム期

初期の単純なデータ処理を行なっていた時期には，計算機システムに恒久的に格納されている情報はなかった．各プログラマは，必要なすべての情報（たとえば，コンパイラ，サブルーチン，データなど）を計算機の所へ運んでこなければならなかった．そして，後で再び使用する目的でシステムが生成した情報

を自分で保存しておかなければならなかった．たとえば，FORTRAN プログラムをコンパイルし，ロードし，実行しようとするプログラマは，FORTRAN コンパイラ，アセンブラ，およびローダのカード・デックを別々に使用した．計算センタではこれらのカード・デックを色分けして，計算機室に用意していた．このようなシステムの各種の情報の項目（すなわちファイル）の分類が最新式のファイル・システムの始まりである．

初期のファイル・システム（システム・テープ）

IBM 7094 FORTRAN モニタ・システム (FMS) などの，初期のバッチ・オペレーティング・システムの多くは，通常“システム・テープ”と名づけられるものに納められた限定されたファイル・システム機能へと発展した．コンパイラやローダなどの各種のシステム・プログラムは磁気テープ（または磁気ディスク）に格納された．プログラマは単に *FORTRAN とか *LOAD とかの制御カードを入れるだけで，オペレーティング・システムはシステム・テープから必要なプログラムを取り出してくれる．これらのルーチンはシステム・テープの中にコア・イメージの形で格納され，オペレーティング・システムはこれら各々の記憶場所を正確に知っている．

ライブラリ・ファイル・システム

次に，頻繁に使用される応用プログラム（たとえば，給与計算，分類／合併パッケージなど）に対してオペレータが手でカードを操作する手間を省くために，これらのプログラム・パッケージは計算機システムに恒久的に格納することが望ましいとされるようになった．プログラムが“システム・テープ”に追加されたり削除されたりする度にオペレーティング・システムを書き直すのは具合が悪いので，より一般的な機構が望まれた．このことから，通常，**ライブラリ管理**と呼ばれるプログラムを導入する必要が生じた．ライブラリ管理プログラムは1台以上の**ライブラリ・テープ**（この中にシステム・テープを含むこともある）の領域割付けを管理する．ライブラリ管理は要求に応じてプログラムを追加したり削除したりし，格納された各々のプログラムの記録場所を示す**指標ディレクトリ**を管理する．

特定のプログラムの実行を要求している制御カードを処理したとき，オペレーティング・システムはライブラリ管理の指標ディレクトリの中を探してプログラムの格納場所を求める．

汎用ファイル・システム

ライブラリ管理によってプログラムの管理と格納の問題は具合よく解決された．しかし，これだけでは他の形の情報——データ・ファイル——の処理は容易にならない．データ・ファイルは，給与や雇用ファイルから人口調査まで，科学的データの表から原始プログラムに至るまで広範囲に分布する．一般化したファイル・システムにおいては，プログラマはこれらの情報を簡単で容易な方法により格納することができる．さらに，プログラマはその情報を後で参照したり変更したりできる．

9.5.2　汎用ファイル・システムの構造

ファイル・システムが処理しなければならない一般的な参照の仕方は：

READ　ファイル〔D〕 ファイル格納場所〔F〕 主記憶装置の番地〔M〕 長さ〔n〕

ここでDはファイルの名前，FとMは各々ファイルと主記憶装置の格納場所である．ファイルは論理的には種々の方法で編成される．最も一般的なファイル編成は**順編成**と**直接編成**である．順編成では，ファイルのすべてのレコードは1つずつ順番に処理される．直接アクセス編成では，ファイルのレコードの一部分が処理され，アクセスは必ずしも順番に行なわなくてもよい．

汎用ファイル・システムのモデル

ファイル・システムは機能的に7つの論理的な段階に分離することができる：

1. アクセス法
2. 論理ファイル・システム
3. 基本ファイル・システム
4. ファイル編成法

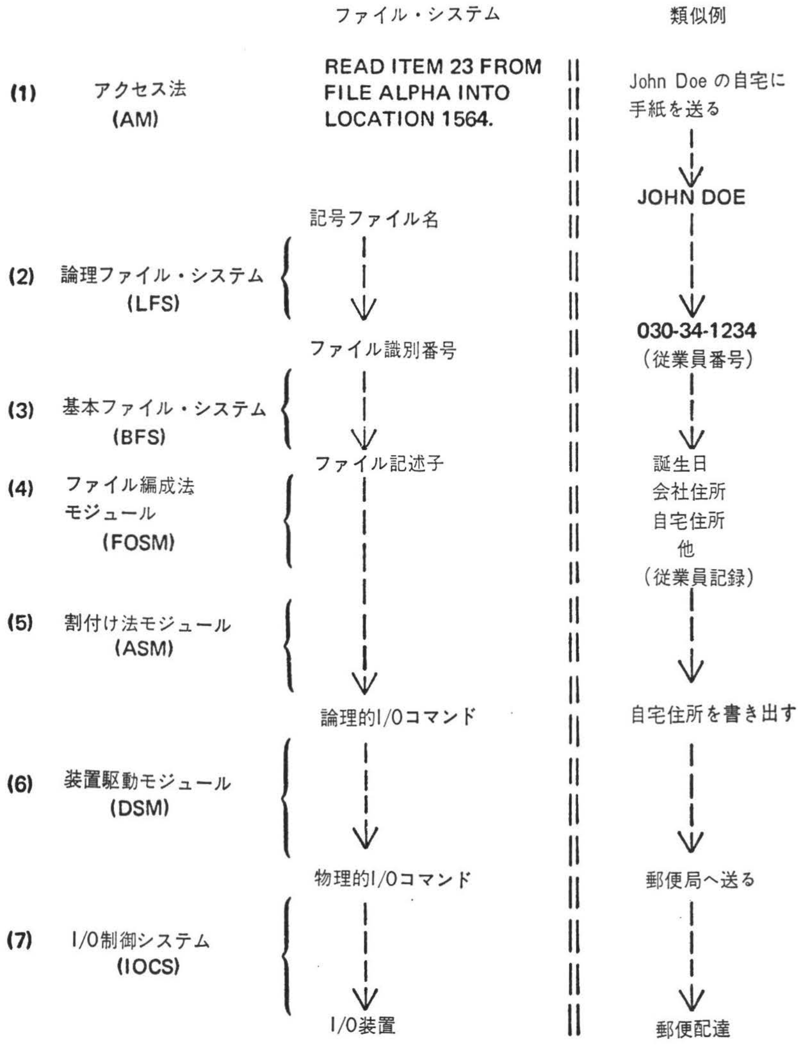


図 9.32　ファイル・システムのモデル

5. 割付け法
6. 装置駆動
7. I/O 制御システム

図 9.32 は汎用ファイル・システムのモデルを示している．たとえば，プログラムが読込みたいファイルを記号名で指定して読込み要求を出すとする．論理ファイル・システムは記号名を受取り，それに対応するファイルの識別番号を求める．基本ファイル・システムはこのファイル識別番号を受取りファイル記述子を求める．ファイル編成法モジュールはファイル記述子を用いて，物理的な番地を決定する．装置駆動モジュールは情報を参照するために（第9.1節で論じたような）物理的なI/Oコマンドを生成する．そしてI/O制御システムは物理的なファイル・コマンドの実行をスケジュールする．本書でファイル・システムの考察と設計に使用するモデルは MIT の Stuart Madnick[3] によって考察されたものである．

9.5.3　ファイル・システムの例

ALPHA という名前のファイルがドラムに格納されているとする．ALPHA は各々長さ 250 バイトの 11 個のレコードから成っている．**図 9.33** は 1000 バ

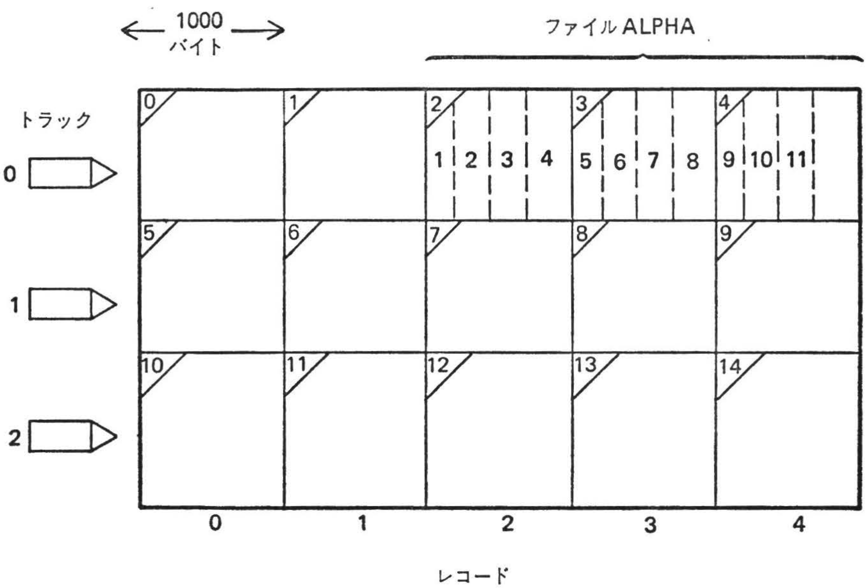


図 9.33　物理的なファイルの記憶

3　"Desingn Strategies for File Systems", Stuart Madnick, MAC-TR-78, MIT, Oct. 1970.

イト長の物理レコードを格納するドラム型の装置を示している．この物理レコードを，ファイルの論理レコードと混乱しないように**ブロック**と呼ぼう．説明のためにドラムは“平らに”してある（磁気ドラム記憶装置の別の見方については図 9.30 参照）．

図 9.33 では，各物理ブロックに番号が打ってある．ブロックの長さは 1000 バイトであるから，各ブロックに 250 バイトの論理レコードを 4 つ格納することができる．ファイル ALPHA の 11 個のレコードが，図に示したように，物理ブロック 2, 3 および 4 に格納されているとする．これらのブロックを ALPHA の**論理ブロック** 0, 1 および 2 と呼ぶことにする．この例では，論理ブロックまたは論理レコードは連続して割付けられている．

要求の例

ファイルを参照しようとするプログラマは，ファイル ALPHA から 6 番目の 250 バイト長の論理レコードを読込み，主記憶装置の場所 BUFF へ複写するために次の PL/I 型の文を書くことができる：

```
READ FILE(ALPHA) RECORD(6) SIZE(250) LOCATION(BUFF)
```

コンパイラはこの READ 文を翻訳してファイル・システムのサブルーチンを呼出すコードを生成する．

上の要求を満たすために簡単なファイル・システムで行なわなければならない動作を考えてみよう：

1. ファイル ALPHA の物理的な格納場所を決定する．
2. レコード 6 を納めている物理ブロックを決定する．
3. 物理ブロックをドラムから主記憶中のバッファに読込む．
4. レコード 6 をブロック・バッファから抽出して記憶場所 BUFF に移す．

最初の動作を行なおうとする前に十分な情報がないことがわかる．ALPHA がどこに格納されているのか．それをどうして求めるのか．このためには各ファイルの名前，長さおよび開始番地を指定する表が必要である．この表はしばしば**ファイル・ディレクトリー**とか**ボリュームの内容表**（Volume Table of Contents：VTOC）とか呼ばれる．後の名前は記憶媒体を“情報の入れ物”として見たための**ボリューム**（volume）という名前からきている．記憶装置とボ

リュームとの区別は，たとえばディスク・パック，テープ・リールなどの**取り外し可能のボリューム**が使用されるとき重要である．これらのボリュームは記憶装置から取り外して，後で別の，場合によっては異なった計算機システムの，(同種の) 記憶装置に装着することができる．

図9.34は VTOC を示している．長さは色々な方法で表現することができる．ここではファイルの内容のバイト数を用いることにする．したがって ALPHA の長さは2750バイト (250バイト長のレコード11個) で，論理ブロック0は実際には物理ブロック2である．

欄	名前	長さ	番地
1	BETA	900	6
2	ALPHA	2750	2
3	GAMMA	2000	7

図9.34　ボリュームの内容表 (VTOC)

いまや READ の処理を**図9.35**に箇条書きに示したように進めることができる．

1. VTOC を探して ALPHA に対する欄を求める (第2欄).
2. レコードの論理バイト位置を (レコード番号－1)×(レコードの長さ) として求める．(6－1)×(250)＝1250
3. 論理バイト位置1250は論理ブロック1のバイト250に対応する．
4. VTOC から 論理ブロック0＝物理ブロック2 であることがわかるので，論理ブロック1＝物理ブロック3 である．
5. 第9.1節に述べた I/O ルーチンを用いて，物理ブロック3が1000バイト長のバッファ領域に読込まれる．
6. 上のステップ3から論理レコード6はバイト250から開始することがわかっている．そこでバイト 250－499 をバッファから番地 BUFF に移す．これで READ 要求の処理が完了する．

図9.35　READ 要求の処理に必要なステップ

問題点の検討

このような簡単なファイル・システムにおいても幾つか検討しなければならない問題点が存在する：

1. どのようにして VTOC の欄が作られ記入されるか．

2. VTOC はどこに格納されるのか.
3. 各要求に対して VTOC 全体を探す必要があるか.

この疑問を1つずつ答えながら，ファイル・システムを段階的に発展させてみよう．IBM のディスク・オペレーティング・システム 360 (DOS/360) のようなシステムでは，プログラマは使用できる記憶領域を覚えていて，次のような制御カードで VTOC を管理しなければならない：

CREATE ALPHA, LENGTH=2750, LOCATION=2

およびDELETE BETA

CREATE コマンドは VTOC に新しい欄を加え，DELETE コマンドは VTOC から欄を取り除く.

VTOC が常時主記憶に置かれていると，高価な主記憶はこの表を格納するのに専用される．しかもこの表はきわめて大きいことがあるので，VTOC をファイルとして扱い，2次記憶装置に入れておきたい．VTOC をファイルとして取り扱うと利用者のファイルは，ファイルと VTOC を納めたボリューム（たとえば，テープ・リール，ディスク・パックなど）を移動することにより容易に他のシステムに移すことができる.

VTOC がファイル・システムの中に格納されると，ファイルを参照しようとするときには VTOC ブロックをロードしたり探索したりしなければならない．VTOC が主記憶に格納されていないとすれば，VTOC を探索するのに大変時間がかかることになる．VTOC は多数の欄を持っていて非常に大きいことがあるが，通常，同時にごく少数のファイルだけが使用される．そこで現在使用中のファイルに対する VTOC の欄を主記憶に複写しておけば，それ以後の探索時間は大幅に減少できる．多くのファイル・システムには2つの特別な要求——特定の VTOC の欄を主記憶に複写するための OPEN および，もはやその VTOC の欄が主記憶に不要であることを示す CLOSE——がある．VTOC の欄の現在の格納場所によってファイルが**開いている**とか**閉じている**とかいう.

9.5.4　汎用ファイル・システムの機能

上に示した簡単なファイル・システムは応用によってはこれでも十分かも知れないが，次のような多くの機能が欠けている：

1. 最小の I/O 動作
2. 論理レコード長と物理ブロック長の関係が自在であること
3. ファイル領域の自動割付け
4. ファイル領域の動的割付け
5. 柔軟な**ファイルの名前づけ**機能（たとえば，リンク，別名，副ディレクトリーなど）

最小の I/O 動作

図 9.35 に示されたアルゴリズムにしたがってファイル ALPHA が順番に処理されるとする．11個の論理レコードをアクセスするのに，各動作が1つの物理ブロックを読込むとして，11回の I/O 動作が必要である．ファイル全体ではただ3つの物理ブロックを使用しているのにすぎないから，余分な動作を省いて I/O 動作の回数を減少できないかと考えたくなるであろう．

I/O 動作の回数は，第 8.2.6.2 節で論じた コンパイラ の 機械に 関係するコード最適化技法に非常によく似た技法を用いることによって減らすことができる．各 I/O 読込み 動作に対して 物理ブロックは主記憶のバッファに複写される．バッファの中に現在どの物理ブロックが入っているかを覚えておき，要求したブロックがすでにバッファの中に あることがわかれば，**図 9.35** のアルゴリズムのステップ5を修正して，I/O 読込み動作を省略することができる．ファイル ALPHA の場合には，レコード1を要求すると物理ブロック2がバッファに複写される．続けてレコード2, 3, および4を要求すると，余分の I/O 動作を必要とせずに バッファ の内容を移動するだけで処理をすませる ことが できる．レコード5を要求するとバッファの中に物理ブロック3が複写される．このようにしてファイル ALPHA の全体は，順番に処理するとき，11回ではなくただの3回の I/O 動作で処理することができる．

この技法はしばしば**ファイル・バッファリング**と呼ばれ，数多くの変形が存

在する．たとえば，開かれたファイルごとに別個のバッファを割当てる場合もあるし，すべてのファイルにただ1つのバッファが共通に使用される場合もある．さらに，ファイルにランダムにアクセスする場合の有害な影響を最小にし，I/Oの同時動作が可能となるように，1つのファイルに複数のバッファを割付けることが望ましいこともある（このことから**ダブル・バッファリング**という言葉がある）．このようなバッファリング技法は第9.2.5節の**デマンド・ページング**で論じたページ置換えアルゴリズムと非常によく似ている．

論理レコード長と物理ブロック長の独立性

前に考えた単純なシステムでは，物理レコードは整数個の論理レコードを含み，余りの部分ができないと考えた．しかし実際にはプログラマは，たとえば長さ13バイト（ブロック長の約数でない）とか，2,415バイト（1つの物理ブロックより長い）とかの任意の長さの論理レコードを処理したいことがある．これらの問題を解決するには種々の方法がある．たとえば，20バイトの論理レコードを使って，これに実際に格納したい13バイトのレコードを入れ，7バイトの余りを残すと，効率は悪いが，実効的には1000バイトブロックに13バイト長の論理レコードを50個格納することができる．

多くの記憶装置では，個々のトラックの書式にかなりの柔軟性がある．たとえば，ある種の装置ではプログラマがトラック書式を1000バイト・ブロック5個（**図9.33**）から300バイト・ブロック15個までの間で，または2000バイト・ブロック3個のいずれかを選択することができる．ブロックの長さに関係なしにすべてのブロックに余分の領域が必要となるので，この書式の関係は通常比例的ではない．

上の解決方法ではプログラマが手操作を行なったりまたは陽に指定する必要がある．読者にとっては，**図9.35**に示された6ステップの手続きを修正して論理レコードが2つの物理ブロックにまたがって入ったり，幾つかのブロックにわたることができるようにすることは容易である．現在のファイル・システムにこの機能の備わっていないものがあまりにも多いのでこの点を強調しておきたい．

ファイル領域の自動割付け

CREATE 制御カードの LOCATION 部で必要とされたようなファイル領域の手操作による割付けは全く不便なものである．第1，プログラマは領域を覚えておかなければならない．これはブロック数が10,000から100,000程度もある通常の記憶装置を使用するときは非常に厄介な仕事である．第2，通常同じ記憶装置を多数の利用者で共用しているから，どの領域が使用できるかどうかを知るには他の人のファイルまで覚えておく必要がある．最後に，次の制御カードを用いたときファイル ALPHA と BETA の領域が重複するが，多くの簡単なファイル・システムでは誤りという表示すらもされない：

CREATE ALPHA , LENGTH = 2750 , LOCATION = 2
CREATE BETA , LENGTH = 900 , LOCATION = 3

特殊の場合にはこのようなことを行ないたいにしても，通常これは誤りである．

上述の理由により，プログラマは単に次のように指定して，ファイル・システムに必要な領域を割付けさせることが望まれる：

CREATE ALPHA , LENGTH = 2750

一般にはプログラマにとってファイルがどこに割付けられようと問題ではない．

システムが領域の管理を行なう技法は多数存在する．1つの簡単な方法は，割当てられていない（空き）ブロックの連続した部分ごとに1つの“模造の”または“空き”の欄を VTOC に作っておくことである．このようにすると，ALPHA に対して3つのブロックが要求されたとき，ファイル・システムは長さが3ブロックであるような VTOC の空き欄を探す．ちょうどこの長さの領域がなければ，より長い空き領域を選び2つの部分に分割する．1つの部分はブロック長が3で，ALPHA に割付けられる．残りの部分は空き領域として割付けられる．**図 9.36** は上の制御カードに対する処理を示している．

VTOC の中には VTOC 自身がどこに格納され，どれだけの領域を使用しているかを示す欄が必要である．このようにするとプログラムは，他のファイル

と全く同様のステップを経て VTOC のレコードを読出したり書込んだりすることができる．たとえば IBM オペレーティング・システム（OS/360）は上述のようなファイル割付け機構を使用している．

ファイル領域の動的割付け

自動的なファイル割付け機構はプログラマの負担を軽減したけれども，なお問題が残っている．ファイル領域の連続的な割付けは主記憶における断片化と同様の現象を発生する（第 9.2 節参照）．たとえば**図 9.36a** では合計 12,000 バイトの空き領域が存在する（3,000 バイトの空き領域 1 つと，9,000 バイトの空き領域 1 つ）が，長さが 10,500 バイトのファイルを作ることができない．IBM 1130 のディスク・モニタ・システムでは実際にファイルを移動してすべての空き領域を連続させ，この問題を解決している（たとえば**図 9.36** のファイル GAMMA はブロック 5 と 6 からブロック 2 と 3 へ移動させる）．しかし，この詰め合わせは非常に時間のかかる操作であり，適当な解決方法ではない．

```
CREATE ALPHA, LENGTH = 2750
CREATE BETA, LENGTH  =  900
```

欄	名前	長さ	番地
1	VTOC	2000	0
2	空き	(3000)	2
3	GAMMA	2000	5
4	空き	(9000)	7
5	・・・		
6	・・・		

(a) CREATEが処理される前

欄	名前	長さ	番地
1	VTOC	2000	0
2	ALPHA	2750	2
3	GAMMA	2000	5
4	BETA	900	7
5	空き	(8000)	8
6	・・・		

(b) CREATEが処理された後

図 9.36 自動的なファイル領域の割付け

多くの場合に前もって所要のファイル容量を予想することができないことがある．たとえば，アセンブラやローダが複写するファイルなどの容量は入力データに関係する（たとえば，原始カードや目的カードの枚数に）．プログラマ

が容量をあまりにも過大に“見積れば”ファイル領域は無駄となるし，過小に“見積れば”プログラムは正しく動作しないであろう．

上述の両方の問題を解決する方法は動的な，非連続なファイル領域割付け法である．この方法では，領域がプログラム実行中にファイルに動的に追加される．また割付けられる物理ブロックは必ずしも連続していなくてもよい．使用できる技法は多数あるが，広く用いられているのは**図 9.37** のようにファイル・マップを使用する方法である．

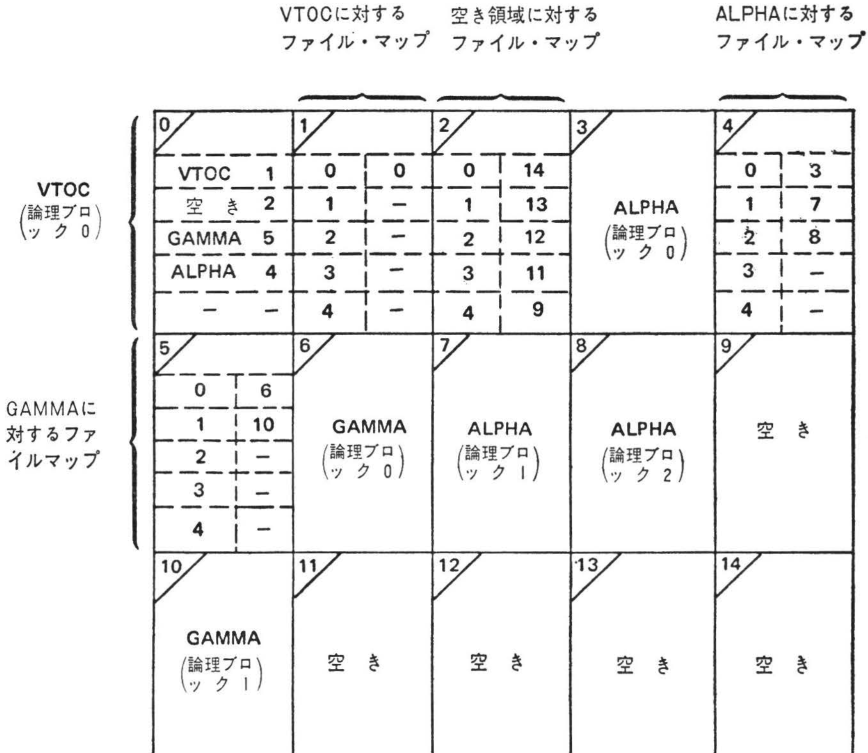


図 9.37　ファイル・マップ構成の例

動的で非連続なファイル領域割付けにおいて，VTOC は**図 9.34** および**図 9.36** と同じ形式を持っているが，場所を示す部分は対応したファイル・マップの物理ブロック番地を指定するのに使用されている．ファイル・マップは，

ファイルの論理ブロック番号と物理ブロック番号との対応を示している．たとえば，ファイル ALPHA のファイル・マップは物理ブロック4に格納されている．ファイル ALPHA に割付けられた3つのブロックは3,7および8である．通常，ファイル・マップはファイルが開かれたときその VTOC 欄と共に主記憶に複写される．この非連続割付けは，READ 要求の処理のステップ4（**図9.35**）を単に“物理ブロック番号＝ファイル・マップ（論理ブロック番号)”と変更するだけでよい．

記入されていないファイル・マップの欄は，対応した物理ブロックがまだ割付けられていないことを示している．この動的割付けの方式を使用するためには制御カードにファイル名だけを規定すればよい（たとえば CREATE ALPHA)．このカードは VTOC の欄とファイル・マップのブロックを設定し，初期化する．これ以外のブロックはこの時点では割付けられない．実行中に WRITE 要求が出されたとき（たとえば WRITE FILE(ALPHA) RECORD(14))，この物理ブロックがまだ割付けられていなければブロックが1つ“空き”ファイル・マップから取り出されて（たとえばブロック9)，ALPHA のファイル・マップに記入され，ファイル ALPHA に割付けられる．次に VTOC の欄はこれらの2つのファイルの新しい長さを示すように修正される．逆にファイルが抹消された場合には，使用していたブロックは“空き”ファイル・マップに登録される．

ファイル・マップを構成すると連続したファイル領域を割付ける必要がなくなり，実行中に動的に割付けることができる．読者は本節でのファイル・マップの機構と第9.2.4節で示したページ・マップの機構との間の類似性に注目してほしい．この類似性については本章の終りで論じる．

柔軟なファイルの名前づけ

記号によるファイル名はプログラマがファイルを記憶したり参照するのにきわめて便利な方法であるが，前述の VTOC 方式では幾つかの欠点がある．多数の利用者がファイルを作ったり使用したりするとき，2人の別のプログラマが別のファイルにたまたま同じ名前をつける（たとえば ALPHA）ことも起こり得る．たとえば1人が次の制御カードを使用すると，他のプログラマのファ

イル ALPHA を消去することになるかも知れない:

DELETE ALPHA /* 古いファイルを消去 */
CREATE ALPHA /* 新しいファイルを作成 */

同様に，READ および WRITE 要求は2人のプログラマが共に同一のファイルを使用することになる．この問題には多数の解決方法が存在する．たとえば名前の“在庫一掃”を行なうなどもその1つである．しかしこの方法は理想的なものに程遠い．

より一般的なシステムでは，2人のプログラマは別々のファイルに同じ名前を使用したい場合がある．一方，2人のプログラマが同一のファイルを別の名前で使用したいこともある．後の場合は，プログラムが完成するまで，2人のプログラマの間でファイル名についての取り決めがなかったような場合に起こる．同様に，1人のプログラマでも1つのファイルを2つ以上の名前で参照できると便利なことがある（この機能はしばしばファイルの**別名**と呼ばれる）．

ここでは柔軟なファイルの名前づけ機能を実現する方法を簡単に示そう．まず，すべてのファイルは名前でなく，番号が割当てられていると仮定する．**図 9.38** はこの状態を示す．

図 9.38 のVTOCは，記号名の部分がなくなっている以外は**図 9.36** の VTOC にきわめて類似している．ファイルは VTOC の欄の番号で参照することがで

VTOC

欄	長さ	番地	
1	500	1	(これはVTOCファイルである)
2	3000	2	(これは“空き”ファイルである)
3	2000	5	(これはGAMMAファイルである)
4	2750	4	(これはALPHAファイルである)
5	900	9	(これはBETAファイルである)
	...	...	

図 9.38　記号名のない VTOC

きる（たとえば，欄番号4は，実は，ファイル ALPHA である）．

システムを簡単にしてしまったので，記号名の機能を新しい方法で回復してみよう．このために**ディレクトリー・ファイル**を導入する．ディレクトリーは正規のデータ・ファイルであるが，記号名と VTOC の欄番号との対応をつけるために使用されるものである．名前づけの混同をなくしたり最小にしたりするために，各プログラマやプログラムの班には別々のディレクトリーが割当てられている．したがって，各プログラマの個人ディレクトリーを規定している**主ディレクトリー**または**根ディレクトリー**を設ける必要がある．

図9.39は簡単な階層的ファイル構造の詳細を示している．VTOC の最初の2つの欄は VTOC 自身と“空き”ファイルを示すものである．VTOC の3番目の欄はつねに主ディレクトリー・ファイルに割当てられる．このファイルはファイル・マップを持つ他のファイルと同様に構成される．

主ディレクトリーは各プログラマごとに1つの欄を持ち，そのプログラマのディレクトリーを指定している（たとえば，JOHN のディレクトリーは VTOC の第8欄で定義されている）．各プログラマのディレクトリーは参照可能のファイルとそれに対応する VTOC の欄との一覧表を持っている．JOHN は2つのファイル ALPHA と BETA を持ち，STUART は4つのファイル ALPHA, BETA, GAMMA および DELTA を持っている．

2人のプログラマのファイル BETA は，実は，同一のものである．このようなファイルをしばしば**リンクされたファイル**と呼ぶ．一方 JOHN のファイル ALPHA は STUART のファイル ALPHA とは別物である．この例にはその他の柔軟性も示されている．STUART のファイル ALPHA と DELTA は，実は，同一のものである．最後に，STUART の GAMMA という名前のファイルは JOHN のファイル ALPHA と同一のものである．読者は図をよく見てこれらの関係を確かめておいてほしい．

上述のような階層的なファイル構造はしばしば**図9.40**で示されたような樹構造で記される．ファイル・システムによっては，プログラマは多層の副ディレクトリーを作ることができる．これらはしばしば個人用ライブラリと呼ばれ，ファイルの構造は何層かになることがある．

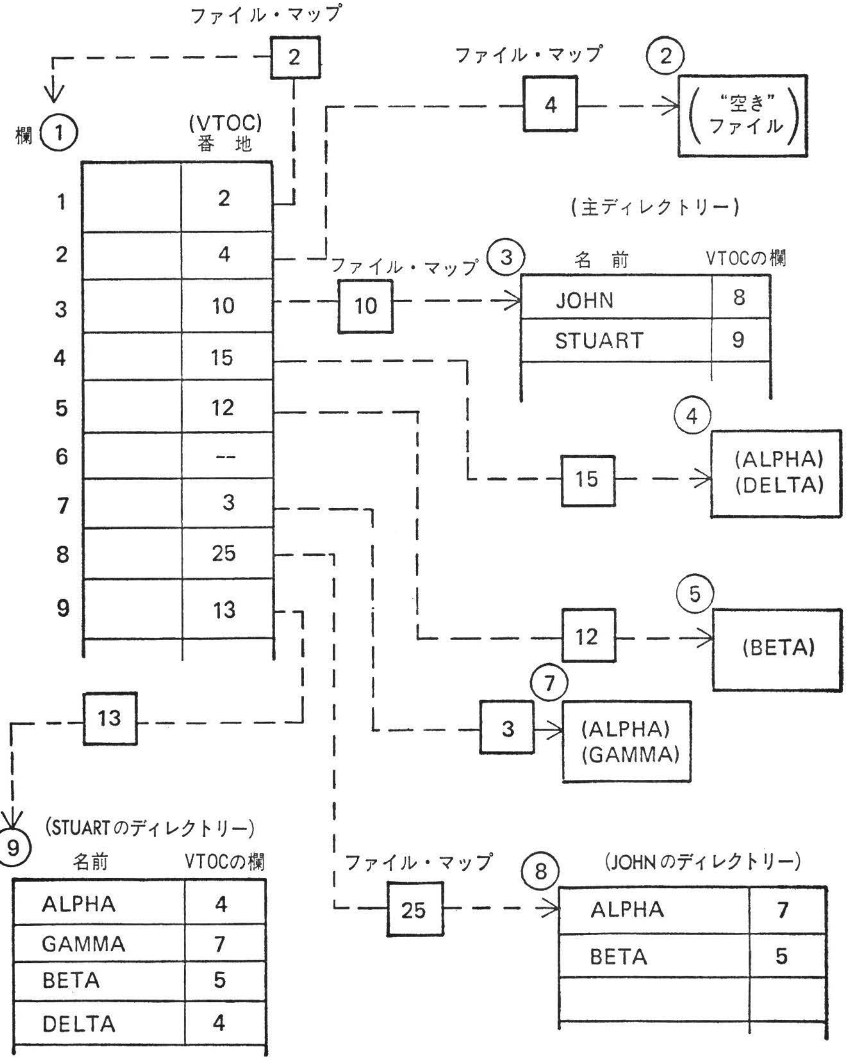


図 9.39　階層的ファイル構造の詳細

9.5.5　汎用ファイル・システムのモデル

第9.5.3節では非常に単純なファイル・システムの設計から論じ始めた．第

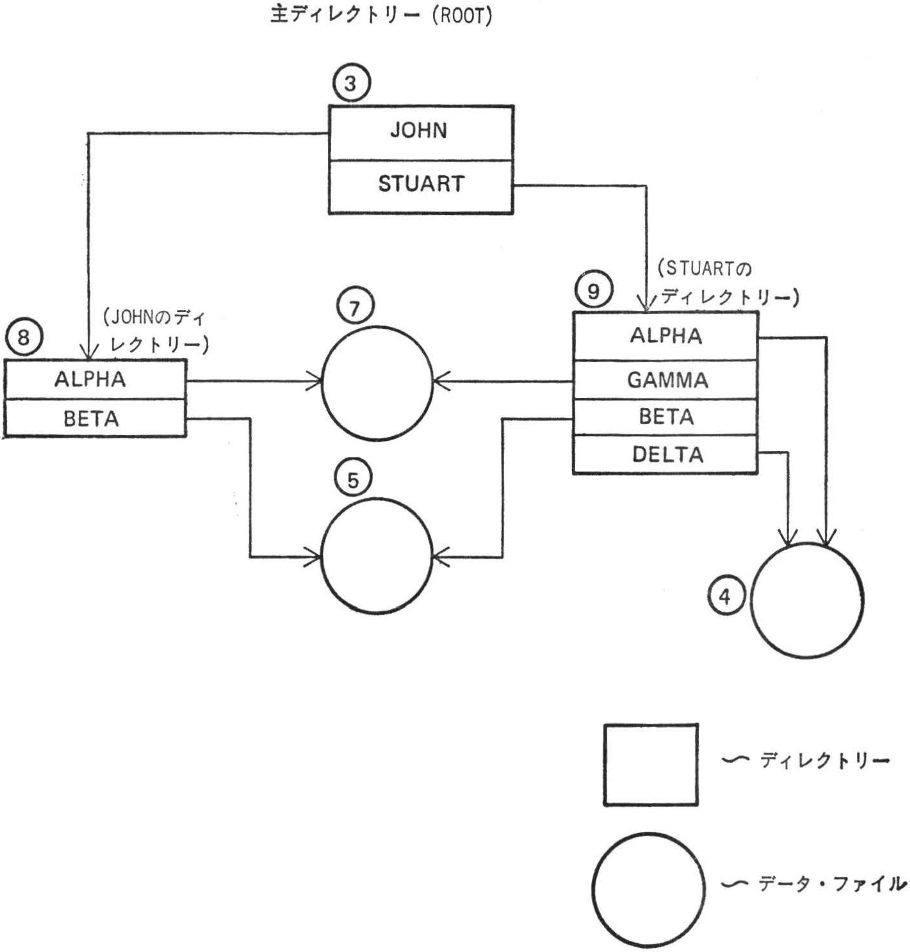


図 9.40 抽象化した階層的ファイル構造

9.5.4節において発展させている間にこの単純な設計は随分複雑になってきた（しかし，複雑になり柔軟性が増したからといって必ずしも大きなオーバヘッドが生じるわけではなく，複雑なファイル・システムは，最適化していない単純なシステムよりもはるかに効率よく働く）.

ここでは**図 9.35** で最初示されたような要求が，9.5.2節で紹介されたような汎用ファイル・システムに対して行なわれるとして，ファイル・システム全

体の構成を考えてみよう．この処理を7ステップに分ける：

1. アクセス法（AM）

   論理レコード番号はファイル中の論理バイト位置に変換される．次に制御は論理ファイル・システムに渡される．

2. 論理ファイル・システム（LFS）

   記号によるファイル名は対応する VTOC の欄番号に変換される．名前がアクティブ名前表（ANT）の中になければ，主ディレクトリーから副ディレクトリーを探して求めなければならない．（これらのディレクトリーは正規のファイルと同じであるから，LFS は副ディレクトリーを読出すために次に述べる基本ファイル・システムを使用することができる．）次に制御は基本ファイル・システムに渡される．

3. 基本ファイル・システム（BFS）

   VTOC の欄番号を用いて，ファイルがすでに開いているか，すなわち，アクティブ・ファイル表（AFT）に欄が作られているかどうかを調べる．なければ VTOC の欄を AFT に複写する．（VTOC は正規のファイルと同様であるから，BFS は VTOC の欄を読出すために，次に述べるファイル編成法モジュールを使用することができる．）

   次に制御はファイル編成法モジュールに渡される．

4. ファイル編成法モジュール（FOSM）

   論理バイト位置は論理ブロック番号とオフセットとに変換される．ファイル・マップを使用して，論理ブロック番号は物理ブロック番号に変換される．

   所要のブロックがすでにバッファの中に納められていれば，論理レコードは直ちに呼び出し者に返却される．所要のブロックがバッファの中になければ，装置駆動モジュールが呼出され，物理ブロックが読出されてから論理レコードが呼出し者に返却される．

   要求が WRITE であり，対応する物理ブロックが割付けられていなければ，割付け法モジュールが呼出されてブロックを割当てる．

5. 割付け法モジュール（ASM）

   “空き”の物理ブロックが“空き”ファイルから取り出され，指定さ

れたファイルのファイル・マップに入れられる.

制御は呼出し者に復帰する.

6. 装置駆動モジュール (DSM)

物理ブロック番号は装置に必要とされる番地の書式に変換される（たとえば，シリンダ番号，トラック番号，物理レコード番号）．そして適当なI/O コマンドが設定される.

次に制御は入力／出力制御システムに渡される.

7. 入力／出力制御システム (IOCS)

IOCS はI/O チャネルの使用を計画し，I/O コマンドの実行を開始さ

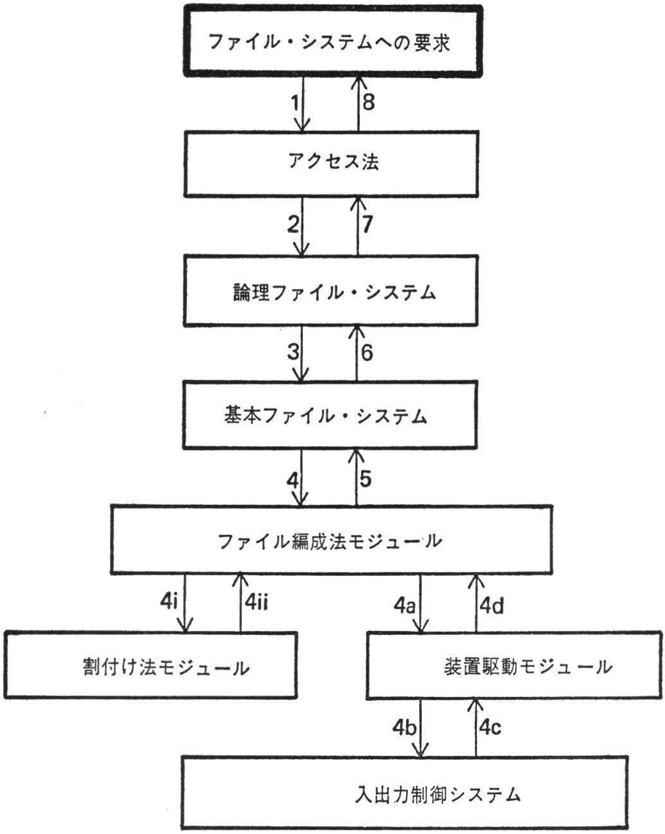


図 9.41 ファイル・システムの構成

せ，割込みを処理する．

I/O が完了すると，制御は呼出し者に復帰する．

図 9.41 はファイル・システムの構成と，要求の処理において起こる呼出しおよび復帰の軌跡を示している．前節で述べた詳細（たとえば，VTOC の書式，ファイル・マップなど）は，個々のシステムによって全く異なっている場合もあるが，最近のファイル・システムの基本構造は**図 9.41** に示したようになっている．

9.5.6　セグメンテーション

第9.2節に述べた（ページングをも行なっている）セグメンテーションは記憶装置管理のみならず情報管理にも適用可能である．汎用ファイル・システムの特徴とセグメント化した記憶システムの特徴とは次のように多くの点で共通したところがある：

1. ファイルとセグメントの概念は両者共論理的な情報の編成を意味する．
2. ファイルもセグメントも任意の大きさを持ち，動的に大きくなり得る．
3. ファイルとセグメントには2次元番地指定が必要である．（たとえば，ファイル名と論理バイト位置に対してセグメント名とオフセット）
4. ボリューム内容表（VTOC）は機能的にディスクリプタ・セグメント（セグメント・マップ）に相当する．
5. ファイル・マップは機能的にページ・マップに相当する．
6. 記憶装置のバッファは機能的にブロック／ページに相当する．

このような比較はさらに続けることができるが，両者の類似性は明らかであろう．セグメンテーションによる情報管理は (1)ファイル・システムとは別の機構として，または (2)ファイル・システムに組込み，1つのセグメンテーション・ファイル・システム機構として，2とおりに利用することができる．HIS 645 システムに組込まれた MULTICS システムでは後者の方法が採用されている．本節ではセグメンテーションの概念を調べ，情報管理を遂行するのにこれがどのように使用できるかを説明する．

情報管理

セグメンテーションの目的

セグメンテーションを行なう目的は次のとおりである：

1. 物理的に**手続きの共有**ができるようにするため．たとえば，平方根のルーチンが２つのプロセスで必要とされるとき，ルーチンの写しを２つ別々に用意することは無駄であるし好ましくない．そこでただ１つの写しを作って共有し，共有できないデータを分離しておきたい．データは，プログラムの実行中に変更され，利用者ごとに異なるので純粋な部分ではない．
2. **ダイナミック・リンキング**ができるようにするため．この機能によってプログラムの実行に必要となるまで，その結合を行なわなくてもよい．さらに，実行時まで呼出されたセグメントの名前を陽に示す必要がない．
3. セグメントが他のセグメントの**コンパイルし直しに不変**となるようにするため．いま，セグメント‘A’がセグメント‘B’の中の変数Xを参照しているとする．Bが後でコンパイルし直され，Xがそのセグメントの中の違う場所に置かれたときでも，呼出しているセグメントAのコンパイルし直しは不要でなければならない．
4. **制御されたアクセス**ができるようにするため．セグメンテーションによってセグメントの保護と論理的な共有が可能となるように組込む必要がある．この機能は互に矛盾するものであるが，最近のシステム利用者から要求されている．
5. **動的にセグメントが成長**できるようにするため．セグメンテーションによって，利用者はセグメントを動的に大きくしたり小さくしたりできる．
6. **２次元番地空間**を用意するため．プログラムは番地空間を２つの演算数——セグメント名とセグメント内のオフセット——によって番地指定する．番地空間は，**図 9.42** に示したように各々が独立に伸びたり縮んだりする多数のセグメントから構成されているとみなすことができる．

上の目的が意味することは次のとおりである：

目的１（手続きの共有）は，共有される純手続きセグメントに対して，別個の共有されず純粋でないリンケージ・セグメントを持つことを意味する．リン

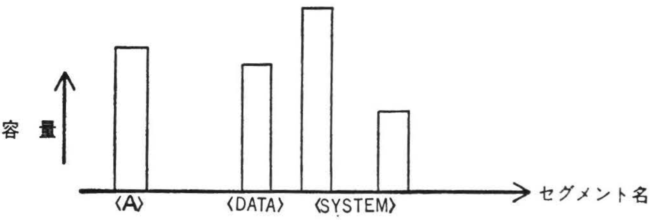


図 9.42　2次元番地空間

ケージ・セグメントは手続きセグメントが行なうすべての外部参照を納めている.

目的2（ダイナミック・リンキング）は，外部参照の結合を実行時まで行なわないことを意味する.

目的3（セグメントは他のセグメントのコンパイルし直しに不変であること）は，すべての手続きセグメントに関して記号表の用意が必要であることを意味する.

また，次に示すセグメンテーションの組込み方法によると目的4，5および6は自動的に満たされることがすぐにわかる.

組込み

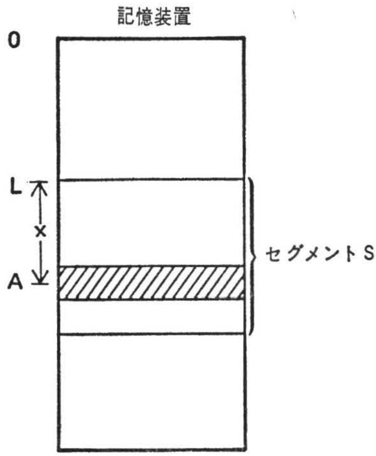
セグメンテーションを行なう計算機システムのハードウェアは，番地参照をセグメント番号とオフセット <S, **x**> として解釈する. **図 9.43** においては参照：<S, **x**> はハードウェアの実際の番地に変換される. ハードウェアはSを表（ディスクリプタ・セグメント）の指標として使用し，セグメントの記憶場所Lを求める. ハードウェアはLにオフセット**x**を加えて番地を求める.

これから論じようとするセグメンテーションの組込みは，MIT, Honeywell および Bell 研究所の共同研究で行なわれ，HIS 645 計算機に組込まれた MULTICS システムに類似したものである.

ここでは**図 9.44** に示されたようなセグメンテーションを行なうと仮定する. 番地空間は**ディスクリプタ・セグメント**によって定義される. ディスクリプタ・セグメントはジョブのすべてのセグメントの情報を納めている. **ディスクリプタ・ベース・レジスタ**（DBR）はディスクリプタ・セグメントを指すように

セグメントSに対するディスクリプタの欄の内容

1. 名　前＝S
2. 長　さ＝N
3. アクセス情報
4. 格納場所＝L



(S, x)→デスクリプタ・セグメントへの指標（ハードウェア）
→⟨d, x⟩→ハードウェア→A＝L＋X

図 9.43　セグメンテーションを行なう計算機におけるハードウェアの番地

設定されていなければならない．リンケージ・セグメントおよびスタック・セグメントは頻繁に参照されるので，通常，これらのセグメントを指すためにレジスタを割当てる．

1つのジョブが実行されているとき，DBR はそのジョブのディスクリプタ・セグメントを指している．新しいジョブが処理されるときには，DBR はそのジョブのディスクリプタ・セグメントを指さねばならない．セグメント番号はディスクリプタ・セグメントの中の指標として使用され，ディスクリプタ・セグメントからセグメントの格納番地を求める．オフセットはその格納番地に

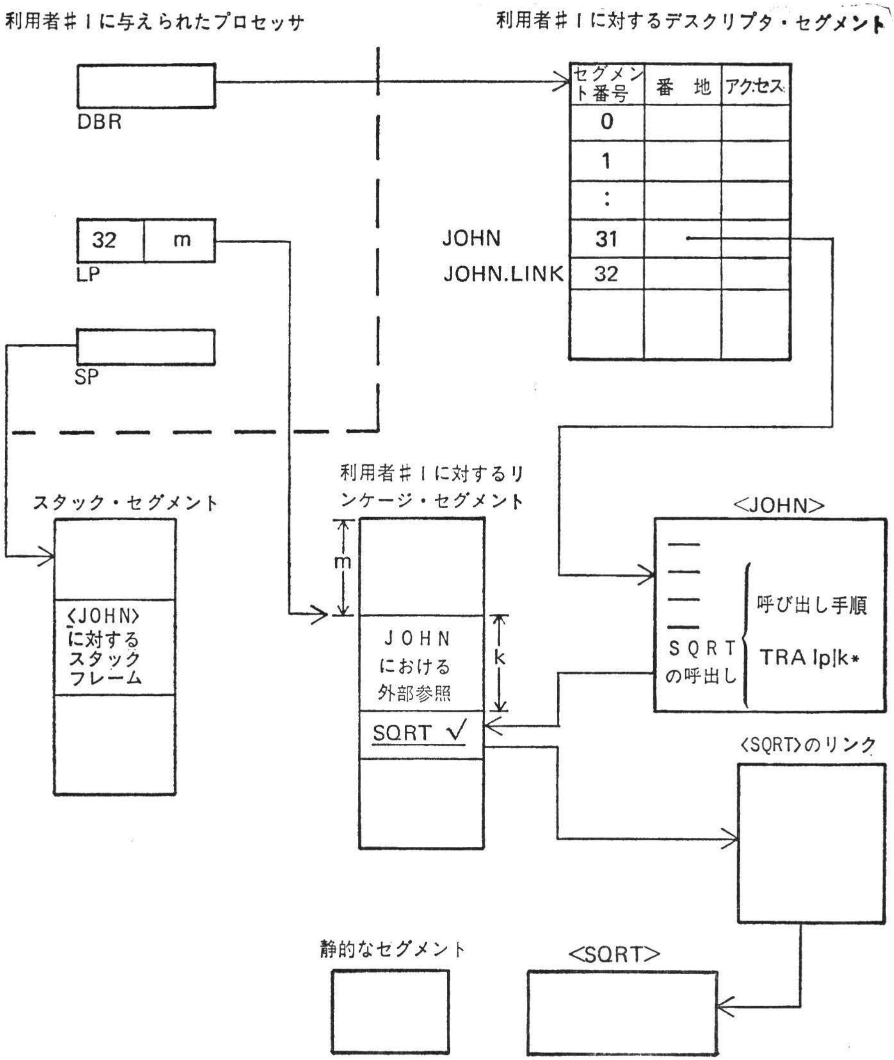


図 9.44　利用者＃1のセグメント化された番地空間

加えられ，目的の番地が求められる．MULTICS システムはページングを行なう記憶方式を用いて組込まれているので，ディスクリプタ・セグメントは主記憶中のセグメントの記憶場所を直接指さずに，セグメントのページ表を指す．

ページングは必ずしもセグメンテーションに必要ではないので，**ページング**については無視することにする．

図 9.44 の例を用いてセグメンテーションの機構を発展させてみよう．名前 JOHN という手続きセグメントがあるとする．JOHN の原始コードは文 CALL SQRT を持っている．コンパイラは CALL 文を翻訳して，制御移行文を最後に置いた呼出し手順を生成する（第8章回帰呼出しを参照）．この文の制御移行先の番地に焦点を合わせてみよう．

呼出しに際しては単純に SQRT ルーチンに制御を渡せばよいと考える人がいるかもしれない．しかし，目的2（ダイナミック・リンキング）によって，コンパイラは直接 SQRT ルーチンに制御を渡す命令を生成することができない．その理由は，このルーチンは実行時に主記憶の中に入っていないかも知れないからである．それで次の方法として，コンパイラがスーパバイザに制御を渡す命令を生成するというのはどうであろうか．スーパバイザは SQRT ルーチンを探してロードし，制御移行命令の番地部を SQRT ルーチンの番地で置換えて，その命令を実行し直す．しかしながら，この方法はうまく働かない．なぜなら JOHN を純粋でない手続きとしてしまうからである（目的1――共有）．したがって，第2のジョブが制御移行命令を実行すると間違った SQRT ルーチンを呼出すことになるかも知れない．たとえば，最初のジョブはシステム・ライブラリの中の SQRT ルーチンを呼出そうとし，一方，第2のジョブは自分自身の SQRT ルーチンを呼出そうとしていることがある．目的1――共有――の要求によって，JOHN は両方のジョブに対して物理的に同一の手続きである．

したがって，コンパイラはスーパバイザへ制御を移す命令を生成せず，JOHN に関連したリンケージ・セグメントに関して間接的に制御を移す命令を生成する．JOHN に関連するリンケージ・セグメントは各利用者ごとに別々のセグメントが作られる．リンケージ・セグメントはきわめて頻繁に使用されるので，レジスタ（LP）を用いてこれを指すようにする．

このようにして，コンパイラは制御移行命令 TRA lp|k* を生成する．* は間接番地指定であることを示し，参照された場所の内容の番地に制御を移す．lp はリンケージ・レジスタ（LP）を示す．k はコンパイラによって計算された

数で，SQRT ルーチンに対応している．SQRT がセグメント JOHN から呼出された最初のルーチンとすると k は1となる．コンパイラは次にリンケージ・セグメントを生成し，その最初の場所に文字列 'SQRT' を記入した語（実際は文字列 'SQRT' を記入した語へのポインタ）を置き，その場所に割出しビットを設定しておく．

実行時にリンケージ・セクションの中のこの場所へ制御が移されると，ハードウェアは割出しビットを検出して割込み，その結果割込み処理ルーチンに制御を移す．

さて元に戻って初めから見直してみよう．CALL SQRT に対応したコードは呼出し手順（すべてのレジスタを保存し，スタック・ポインタを更新する）であり，制御移行命令で終了する．この命令はリンケージ・レジスタを用いて間接的に分岐する．リンケージ・レジスタは2次元番地——セグメント番号（32）とオフセット（m）——から成っている．ハードウェアは32をセグメント表の指標として用いてリンケージ・セグメントの場所を求める．次にその番地に m を加えて JOHN に関連したリンケージ・セグメントのセクションの番地を求める．ハードウェアはその番地に k を加えて，リンケージ・セクションの中の SQRT に関連した番地を求める．ハードウェアは，最初にこの場所を参照するときに，割出しビットがオンであることを検出する．この結果，制御は管理システムに移される．この時点で管理システムは SQRT セグメントをファイル・システムの中で探さなければならない．

特に，セグメントなし割出しが起こると，制御はセグメント割出しハンドラに渡されて，適当なディスクリプタ・セグメントの中にセグメント属性が格納される．セグメント割出しハンドラは，本書のモデルでは基本ファイル・システムに相当する．セグメント SQRT を探すに当って，主記憶の中にあるセグメント表を引き，セグメントがすでにロードされているか否かを調べる．表になければ新たに SQRT がロードされる．セグメントが主記憶に入ると，セグメント割出しハンドラはディスクリプタ・セグメントの中に，セグメント番号，格納場所，およびジョブ番号1に関するアクセス権を記入する．SQRT セグメントが別のジョブに対するセグメントとしてロードされるとき，別のアクセス権を持つこともある．次にセグメント割出しハンドラはセグメント割出しビッ

トを SQRT の（リンケージ）セグメントの番地で置換える．

SQRT に入ると，さらに，たとえば CALL TANGENT のように，他を呼出すことがあるので，LP を SQRT のリンケージ・セグメントを指すようにしなければならない[註]．この TANGENT 呼出しは同様な呼出し手順で最後の命令は TRA lp|k* にコンパイルされる．LP の値を変更せずにこの命令が実行されると，JOHN に関連したリンケージ・セグメントを通じて制御が渡されるので，正しい制御移行ができない．したがって，SQRT に入る前に LP が SQRT に関連したリンケージ・セグメントのセクションを指すように変更しておかなければならない．この方法の１つとして次のような手順が実行される．JOHN から制御を渡すときに，JOHN のリンケージ・セグメントを通じて SQRT のリンケージ・セグメントに入るようにする．SQRT のリンケージ・セグメントでは LP レジスタを SQRT のリンケージ・セグメント（すなわち，自分自身）を指すようにしてから SQRT ルーチンに制御を移す．

セグメンテーションの組込みの詳細を長々と論じるのは本書の目的とするところでない．セグメント単位の保護は，コアを参照する度にハードウェアがディスクリプタ・セグメントを検査しなければならないことから自動的に保証される．保護の機能を制御するためにディスクリプタ・セグメントの中に“保護ビット”を加えることができる．

上に概説した方式は多重処理システムに対しても使用することができる．

9.5.7 MULTICS のファイル・システムと一般的なモデル

図 9.45 は MULTICS のセグメンテーションと汎用ファイル・システムのモデルとの対応を示したものである．**図 9.45** においてファイル・システムを働かせるために使用されるコマンドは，外部で定義されたセグメントに情報を格納するコマンドである．MULTICS においては，この命令は“あるリンケージ・セクションを通じて間接的に格納せよ”というコードを生成する．ここではセグメンテーションにページングを組合わせている．

訳註：セグメント割出しハンドラの処理およびリンケージ・ポインタの設定については原文の記述では意味が通じない．訳著の責任で本書のように修正しておいた．なお，最近の MULTICS ではリンケージ・ポインタの設定は手続き本体の中で行なわれるよう変更されている．

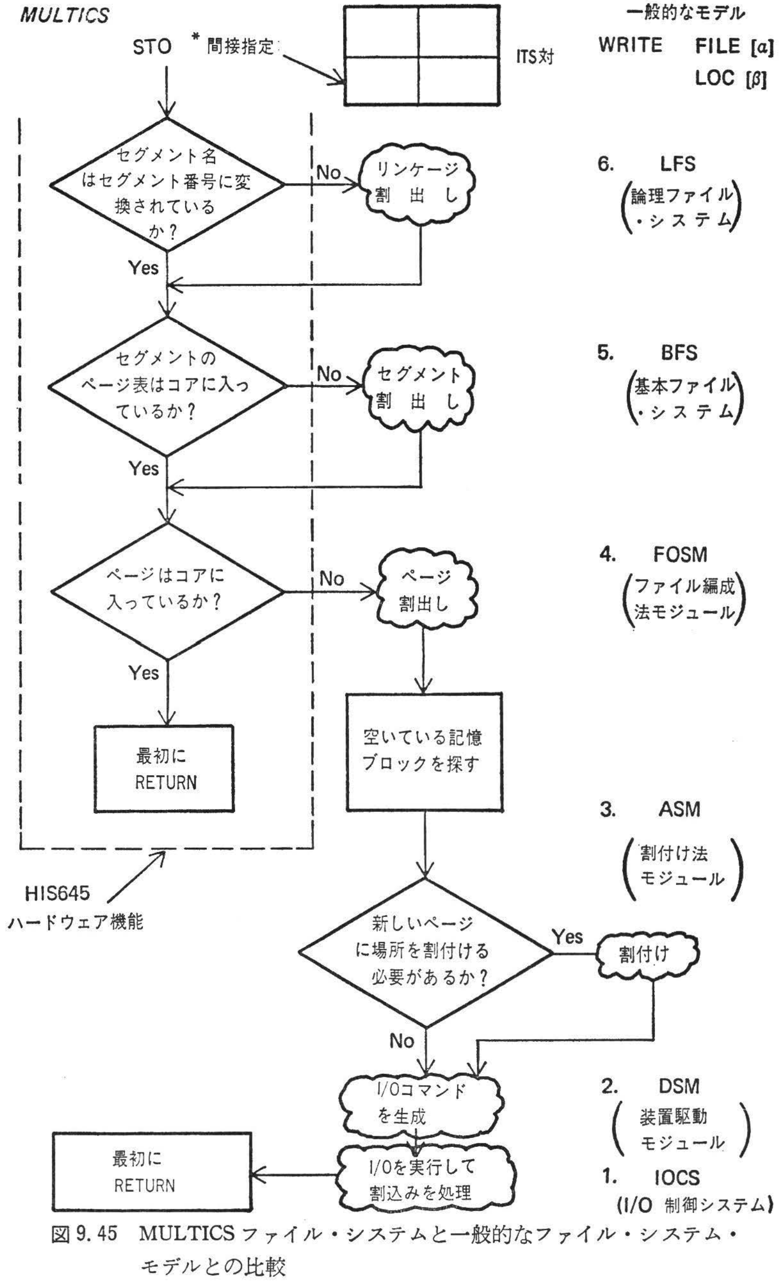


図9.45　MULTICSファイル・システムと一般的なファイル・システム・モデルとの比較

セグメンテーションに関連したオーバヘッドが存在する．記憶装置の中のデータ項目に普通にアクセスする度に，ハードウェアはディスクリプタ・セグメントをアクセスし，次に情報の場所をアクセスしなければならない（実際には，ディスクリプタ・セグメント，ページ表，記憶場所の順にアクセスする）．このオーバヘッドは，連想記憶装置を用いたり，幾つかのセグメントを結合しておいて外部参照を避けることによって減少させることができる．この結合は頻繁に実行され，また完全にデバッグされたセグメントについて行なうことができる．一般に，セグメンテーションは多くの利点を有し，将来の多くの計算機システムに用いられるであろう概念である．

9.6 ま と め

本章ではオペレーティング・システムの種々の面と，オペレーティング・システムの幾つかのモジュールの設計について論じてきた．オペレーティング・システムの特性としては，本書で述べてきたシステムの構成要素に密接に関連した面に焦点を合わせた．そして，オペレーティング・システムを記憶装置割付け，処理装置割付け，装置割付け，および情報割付けに至る広範囲の機能について統一的に論じることのできる資源割付けの観点から見た．これらのモジュールの多くについては詳細な設計を行なわなかったが，読者には各種の変形の可能性を示すように試みた．プロセッサの状態のモデルは広い範囲の計算機に適用できる．小規模なリアル・タイム・システムから本書で述べた大規模システムまでこの考えを適用してみてほしい．同様にファイル・システムについて与えられたモデルは，設計の面でも研究の面でも広範囲のファイル・システムに適用できるであろう．

資源管理の議論を通じて，システムがこれら資源の管理に果たす役割りを強調した．このことから利用者自身は，オペレーティング・システムと同様に，自分の資源を管理できるかという問題が生じるであろう．オペレーティング・システムは，多重プログラミング・システム全体の状態を知っていてこれを最適化することができ，全体的な効率を向上させるような決定を動的に行なうことができる．多重プログラミング・システムにおいて，全体的に最適なシステ

ムの運用法と，１つのジョブが単独で実行されるとして最適化する方法とはめったに一致しない．

演習問題 [4]

第 1 部

1. a. チャネルとは何か．
   b. チャネルの役割りは何か．
   c. チャネル状態語とプログラム状態語の機能は何か．類似点は何か．これらはチャネルとCPUの間の通信にどのような役割りを果たすか．
   d. チャネルはI/O動作が終了した時刻およびその原因をCPUに知らせることができるか．どのようにして行なうか．
2. I/Oチャネルがなく，命令“WRITE CHARACTER,”“OUTPUT I/O COMMAND,”“TEST I/O STATUS,”を持つ小さな計算機を考えよう．計算機はこれらの命令を用いてテレタイプ型の装置に一時に１文字だけ，毎秒10字の速度で出力することができる．入出力動作はプロセッサ自身の制御によって行なわれる．このようなシステムにおける割込みの意義は何か．
3. *1600_{16} 番地から開始する次のI/Oのチャネル・プログラムが1403型印刷機で実行されるとする．このI/Oプログラムを調べて，何が印刷されるかを述べよ．

番地		
1600	8B 000000	40 00 0000
1608	09 001638	40 00 000A
1610	11 001638	40 00 000A
1618	11 001642	80 00 000B
1620	00 001700	40 00 0004

4 ＊はIBMマニュアルの使用を要する問題であることを示す．

1628	09 001638 40 00 000A
1630	89 001638 00 00 000A
1638	5C 5C5C5C 5C 5C 5C5C
1640	5C 5CD1D6 C2 40 D5E4
1648	D4 C2C5D9 40 00 0000
……	
1700	F6 F2F5F1 00 00 0000

4. **図 9.9** において，番地 38，40，48，および 78 は重要である．これらの場所を変更するのは何か（I/O チャンネル・プログラムか，利用者プログラムか）．変更する理由は何か．いつ変更されるか．

5. I/O チャンネルのプログラミングは機械語で CPU のプログラミングを行なうのと同じである．CPU プログラムを書く過程を容易にするために，アセンブラが発明された．第 3 章ではアセンブラの設計を論じている．
   a. I/O 機械命令に対して，“アセンブラ語”を作ることができるか．
   b. IBM 360 に対して I/O アセンブラ語を定めよ．すなわち，各命令に対して記号名を与えよ．
   c. この言語を処理するために第 3 章の基本アセンブラに加えなければならない変更は何か．
   d. IBM システムでは，利用者に“I/O アセンブラ語”のプログラムを書かせるより，I/O 要求を処理するシステム・マクロを使用するようにしている．アセンブラよりこの方法の方がよい点をあげよ．

6. IBM 360 で“A 1,1317”という命令は，語境界を番地指定していないので，割込みを起こす．このような番地指定誤りを受理して訂正するような IBM 360 用の割込みルーチンを作れ．すなわち IBM 360 が，IBM 370 と同じように，データが整列していなくても正しく働くようにせよ．

第 2 部

7. 次に示すコンパイラとアセンブラのデータ・ベースの内でどれが純粋で，どれが純粋でないか述べよ（純粋とは自分自身を修飾しないものをいう）．1

～2文で説明せよ．

a.　記号表

b.　変形則

c.　コード

d.　マクロ表

e.　命令コード表

f.　行列

8.　オペレーティング・システムの記憶装置管理における断片とは何か．また それはどのようにして解決することができるか．

9.　a.　ページとセグメントの相違点を3つ述べよ．

b.　セグメンテーションなしでページングすることは可能か．その逆はどうか．

10.　純手続きを定義し，これを用いることの2つの利点を述べよ．

11.　システムを設計するときに，システム・アナリストは熱心なあまり，考えつく限り複雑な記憶装置管理を適用しようとすることがよくある．どのような場合に，高度の技法の組込みがシステムの能率向上に役に立たないか，または害にさえなるかを述べよ．

12.　記憶装置管理の節では，記憶装置の資源管理をもっと柔軟にする幾つかの計算機構成法を示した．次に示す記憶装置のハードウェア構成によって可能となる機能と，記憶割付け方式を説明せよ．他のシステムと比較して利害得失をあげよ．

a.　単一連続割付け

b.　再配置型分割割付け

c.　単純ページング

d.　デマンド・ページングを行なうセグメンテーション

13.　a.　利用者の番地空間を定義せよ．

b.　番地空間は次に示す割付け法により割付けられた主記憶とどのように対応するか．

1）　単一連続割付け

2）　再配置型分割割付け

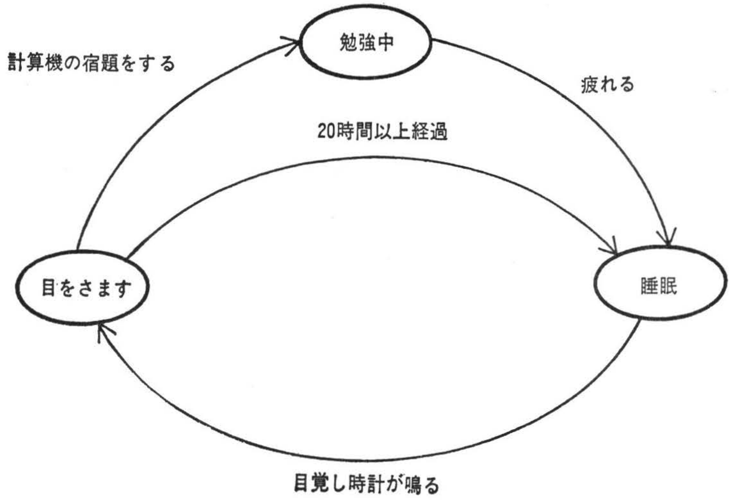
3) デマンド・ページングを行なうセグメンテーション

第 3 部

14. 多重処理システムでは，プロセッサとプロセスとの間の一対一の対応を必要とせず，プロセスに任意のプロセッサを割付けることができる．その理由と方法を述べよ．

15. 実行待ち，実行中および閉塞中の状態のほかに，**停止状態**を導入する．停止状態は大規模なシステムでは重要である．大規模システムでは，１人の利用者が休むと（たとえば，次に何をしようかと考えているときなど），主記憶領域を解放することが望まれる．停止状態とは利用者プロセスが現在２次記憶に置かれているがいつでも実行を再開でき，依然としてスケジューラに登録されている状態を示す．ジョブの休止のほかに停止状態になるのは，その利用者の優先順位が非常に低く，高順位の利用者に主記憶領域を空けるために**ロール・アウト**される場合である．停止状態は，いつでも実行できるが，主記憶の中に入っていないので実行待ち状態ではない．

a. 上述の４つの状態に対して状態遷移図を描け．図における状態の遷移の理由を述べよ．遷移図の例として，読者の日常の行動を示した次の



状態図を考えよ．読者の作る図は**図 9.20** と同じようにせよ．

b. 低い経費で容量を増せる2次記憶に比べて，コア記憶装置は高価で，限定されている．ページングが可能とすると，プログラムを実行するのにプログラム全体をロードする必要は必ずしもない．ページ割出しハンドラは要求に応じて2次記憶装置（ドラム，ディスク）から不足しているページを取り出すことができる．このような動作に要する時間は他のI/O装置に比べわずかである．したがって，前述のプロセスの状態の1つを2つに分けることが適当と考えられる．どれを分けるのか．その理由は何か．**ページ待ち**を新しい状態とすると，分割してできたもう一方の状態に対して旧名が使用される．状態遷移図を書き直せ．この状態遷移を定義せよ．この新しい状態を区別することの利点は何か．

c. 実時間プロセスとは実時間内に処理が行なわれる必要のあるプロセスをいう．すなわち，その処理が実行される時間が非常に重要なのであって，その時間内に完了しないときその処理は価値がない．たとえば，原子炉を制御するには，爆発や運転停止を防止したりするために迅速な応答が必要である．したがって，実時間処理には“起こされた”ときにシステムの実行権を先取りできるようきわめて高い優先順位が与えられている．実時間プロセスに対して遷移図を変更しなければならないところがあるか．

d. システムに接続できるI/O装置には多くの種類がある．たとえば，出力メッセージは後刻印刷するためにテープに送られることもあるし，オンラインの応答としてテレタイプに送られることもある．ある状態をTTY 状態 と TAPE 状態に分割するべき理由があるか．あるとすればどの状態を分割するのか．なぜか．状態図を書き直せ．

16. 競争と手詰まりを区別せよ．これらはどのようにして起こるか．どのようにしたらこれを防止したり，起こってからの回復処理ができるか．

17. プロセッサ間の通信はどのようなとき必要か．

第 4 部

18. スプーリングとは何か．すべての種類のI/Oに適用可能か．

19. I/O トラフィック制御の機能について述べよ．

20. 順アクセス装置を使用するのが適当であると考えられる応用例を示せ．直接アクセス装置の使用を必要とする応用例をあげよ．各答えを簡潔に説明せよ．

第 5 部

21. OPEN および CLOSE 指令の目的は何か． これらを用いるとなぜシステムの効率が向上するのか．

22. どのような理由で，セグメント化，およびページ化された環境は，ファイル管理に関連するのか．

23. 読者は非常に単純な計算機に対してファイル・システムを開発している．この計算機のI/O動作はアキュムレータに1文字を入れたり出したりすることだけである．レジスタ EXT はI/O装置の機番を左半語に置いて指定し，右半語には指定されたI/O装置の文字位置を指定する．I/O 命令の例は，

READ (アキュムレータに入れる) (EXT により指定された装置から)

a. 第9.5節にて与えられたようなファイル・システムをこの計算機に作ることは可能か．説明せよ．

b. 上述のI/O転送システムに最も影響のあるシステムのモジュールは何か．上述した以外の条件についてはレジスタを2つだけ持つ IBM 360 型の計算機であると仮定する．

c. このようなファイル・システムは次の要求をどのように処理するか概略を述べよ．

WRITE FILE(TAPE) FROM(CARD)

ここでは TAPE はテープ・ファイルの記号名，CARD は 80 文字のレコードとする．どのモジュールがどの機能を果たさねばならないかを示せ．

24. a. 第4章の計算機固有の問題は**連続使用可能**なプログラムの例である．連続使用可能なプログラムとは何か．“連続使用可能なプログラム”と“純手続き”との類似点と相違点を述べよ．

b. タイム・シェアリング・システムの動作にセグメンテーションやページングは本質的な要素か．読者の考えを説明せよ．

c. セグメンテーションのどの機能が動的記憶域割付けを可能とするか．

d. セグメンテーションにより，どのようにして保護を行なうことができるか．

25. 次にあげたのはオペレーティング・システムに含まれる典型的なハードウェア／フソトウェアのモジュールである．一覧表に続く問に答えよ．

ファイル・システム
ジョブ・スケジューラ
ファイル記述子のファイル
ファイル識別子のファイル
ドラム
ディスク
ライン・プリンタ
コンソール・タイプライタ
主記憶装置（コア）
拡張記憶装置（プラスチック・カード）
カード読取り機
ディスクリプタ・ベース・レジスタ（DBR）
リンケージ・ポインタ（LP）
引数ポインタ（AP）
スタック・ポインタ（SP）
テープ
再配置レジスタ

手続き用ベース・レジスタ（PBR）

I/O チャネル

a. 上のモジュールを (1)主記憶，(2) 2 次記憶に区分せよ．

次の各問については各モジュールが必要とされる理由と，それが他のモジュールとどのように結合されるかを述べよ．ブロック図を描くことが望ましい．

b. (1)セグメンテーション，(2)ページング の各々に必須のモジュールはどれか．

c. ダイナミック・リンキングの組込みに必要とされるモジュールはどれか．

d. 利用者プログラムを主記憶にロードするのに使用されるモジュールはどれか．

e. 次のファイル・システムの各機能の設計で必要と考えられるのはどのモジュールか．

1) アクセス法（AM）

2) 論理ファイル・システム（LFS）

3) 基本ファイル・システム（BFS）

4) ファイル編成法モジュール（FOSM）

5) 装置駆動モジュール（DSM）

6) 割付け法モジュール（ASM）

7) 入出力制御システム（IOCS）

26. 手続きセグメントが別のセグメント（リンケージ・セグメント）に置かれたリンクを使用しなければならない理由は何か．

27. 1つの手続きに対するリンケージ・セグメントはどのようにして番地指定されるか．また，手続きが呼出されたときどのようにしてその番地が設定されるか．

28. **図 9.37** で示したファイル・システムを考える．次の READ 要求を満たすために必要となるステップを示せ．（**図 9.35** と同様な形式で答えよ．OPEN や CLOSE の使用は無視してよい．）

a. READ FILE(ALPHA) RECORD(4) SIZE(55) LOCATION(BUFF)；

b. READ FILE(ALPHA) RECORD(19) SIZE(55) LOCATION(BUFF);

29. a. HIS 645 および IBM 360 のハードウェアのどの特徴が純手続きを組込むのに有用か.

b. セグメンテーションは保護の問題を解決するのによい方法か. その理由は何か.

30. a. DLL（直接結合型ローダ）の説明において述べたように, プログラムをローディングするには少なくとも2つの処理が必要である. 結合の処理は外部参照を解決し, ローディングの処理は物理的にプログラムを主記憶の中に入れる. MULTICS ではこれらの処理がどのような順序で行なわれるか. 第2の処理は（DLL の場合のように）第1の処理の直後に行なわれるのか. その理由は何か.

b. MULTICS のようにセグメンテンーションを行なうシステムに対してアセンブラは RLD カードを生成する必要があるか. 読者の答えを説明せよ. MULTICS でも IBM 360 と同様の番地定数を持っているものとする.

c. MULTICS ではファイルとセグメントは同じ概念を示している. すなわち, 情報の論理的な単位——具体的には語——の並べられた集合である. 1つのセグメント（ファイル）は MULTICS システムで作られたセグメント本体, 名前, および記述子から構成される. 記述子の目的は何か. たとえば, どのような情報が記述子に置かれ, MULTICS でいかに使用されるか.

31. **図 9.44** はダイナミック・リンキング・ローダ方式を示している. プロセスが手続き JOHN を実行しているとき, CALL SQRT が起こったとする.

a. どの手続きが値 m を設定するか（たとえば, ローダ, リンカ, トラフィック制御, JOHN, JOHN のリンケージ・セグメント, SQRT, その他).

b. LP の内容を設定するのはどれか.

c. k の値を設定するのはどれか.

d. 直接 SQRT に移行せず, SQRT のリンケージ・セグメントに制御を移している理由は何か.

32. HIS 645 で使用されている2次元番地指定法によると，DBR の内容を変更するだけですべての名前空間を“配置し直す”ことが可能である．

セグメント k への参照はすべて，セグメントの開始番地として C(DBR)+k 番地の内容を用いて行なうことができる．DBR の内容が変更されると，C(DBR)+k は前と異なり，セグメント k の開始番地を求めるために別の番地が調べられることとなる．

2次元番地指定法を組込むのに，DBR を用いずに行なう（ハードウェア／ソフトウェアの）方法が幾つかある．1つの方法は幾つかのレジスタを用意し，各レジスタには対応したセグメントの番地を納めておく方法である．このような方法と HIS 645 で用いられている方法とを比較して利点と欠点を論じよ．

MULTICS システムでは，ある利用者のプロセスを実行するのにそのプロセスに関連したディスクリプタ・セグメントの開始番地を DBR に設定すればよい．このようにすると，同時に幾人かの利用者のディスクリプタ・セグメントを主記憶の中に入れておくことができる．プロセスを切換える手続き（SWAP_DBR）はすべてのプロセスの名前空間の中で**同じセグメント番号**を持っている（すなわち，ディスクリプタ・セグメント中のオフセットはすべてのプロセスについて等しく，$k_A = k_B$）．したがって，DBR の内容を変更する1つの命令を実行することによって，プロセスを切換えることができる．

切換え手続き（SWAP_DBR）がプロセスごとに違ったセグメント番号を持っているとしたらどれ位処理が複雑になるか述べよ．

33. HTS 会社（Humble Time-Sharing Co. おそまつな TSS 会社の意味）は，同時に約50人の遠隔利用者（ジョブ）を処理する小さな TSS 会社である．HTS は標準構成の“幸運Ｉ型”実時間多重プログラミング・タイム・シェアリング汎用オペレーティング・システム（ページングとセグメンテーションつき）を使用している．主任販売員 Frank Fasttalker を雇った後で HTS では 100 人の同時利用者が生じ，適当な仕事量を保持するのに問題が起こっている．HTS の計算機部の Harry Moredough は仕事量を増加させるのに次の4つの変更を提案している：

a.　主記憶を買い足す.

b.　新しく高速の2型 CPU を購入する.

c.　古い“象A型”ドラムを新しくて高速の“カバB型”のドラムに置換える.

d.　効率の良い情報管理機能を持つ“幸運Ⅱ型”オペレーティング・システムに切換える.

これらの各々の提案に対して，(1)大きく影響する要素，(2)ほとんど効率に影響しない要素（例えば，ジョブの混合，CPU 制限ジョブなど）について述べよ.

34. 第1章の問4および問5は OS 360 の流れ図を書けという問題であった. OS 360 を組込んでみよ.

第 10 章

参考文献と学習の進め方

1冊の教科書で計算機システムのすべての分野を網羅するのは不可能である．本章では関連した本や論文をあげる．参考文献は，本書の内容の順に，主題毎にまとめてある．

一般的な内容の参考文献

Gear, C. William: *Computer Organization and Programming*, McGraw-Hill, New York, 1969.

Hartman, Philip H., and David H. Owens: *How To Write Software Specifications; Proceedings, 1967 Fall Joint Computer Conference,** pp. 779-790.

Knuth, Donald E.: *The Art of Computer Programming*, Addison-Wesley, Reading, Mass. (seven volumes, two already published).

Rosen, Saul: *Programming Systems and Languages*, McGraw-Hill, New York, 1967.

Wegner, Peter: *Programming Languages, Information Structure and Machine Organization*, McGraw-Hill, New York, 1968.

計算機の構造，アセンブラ語，IBM 360／370

幾つかのIBMマニュアルを示す．読者は別の計算機の構造，並列処理，プロセッサの機能部分の柔軟な割付け，スタックを持つ計算機，バッファ記憶装置，または連想記憶装置などを扱った本を読むのもよい．本書に含めなかったもので重要な領域の1つは，マイクロ・プログラミングである．

---

* Communications of the ACM, Fall Joint Computer Conference, および Spring Joint Computer Conference は各々 CACM, FJCC, および SJCC と略記する．同様に JACM は the Journal of the Association for Computing Machinery を，AFIP は American Federation of Information Processing を示す．

Germain, Clarence B.: *Programming the IBM 360*, Prentice-Hall, Englewood Cliffs, N.J., 1967.
IBM System/360 Principles of Operation, Form No. A 22-6821-7.
A Programmers' Introduction to the IBM System/360 Architecture, Instructions, and Assembler Language, C20-1646-4.
IBM System/360 Operating System Assembler Language, C28-6514-6.

アセンブラ

Corbató 等の教科書が優れた入門書である．本章の一般の部の Wegner をも参照せよ．

Batson, Alan: "The Organization of Symbol Tables," *CACM*, vol. 8, No. 2, February, 1965, pp. 111-112.
Corbató, F. J., J. W. Poduska, and J. H. Saltzer: *Advanced Computer Programming*, MIT Press, Cambridge, Mass., 1963.

表 の 管 理

本書では基本的な技法を論じた．参考文献は他のアルゴリズムを発展させている．取り扱わなかった題目はリスト表現とその処理である．LISP はリスト処理言語である．

Madnick, Stuart: "String Processing Techniques"; *CACM*, vol. 10, no. 7, July 1967, pp. 420-424.

探索法と分類法

CACM, vol. 6, no. 5, May, 1963, ACM Sort Symposium. The whole issue is on sorting. Especially good articles are;
Gotlieb, C.C.: "Sorting on Computers."
Hibbard, Thomas N.: "An Empirical Study of Minimal Storage Sorting."
Hall, Michael H.: "A Method of Comparing the Time Requirements of Sorting Methods."
Buchholz, Werner: "File Organization and Addressing," *IBM Systems Journal*, vol. 2, June 1963, pp. 86-111.
Hamming, R.W.: *Numerical Methods for Scientists and Engineers*, McGraw-Hill, New York, 1962.
Iverson, Kenneth E.: *A Programming Language*, Wiley, New York, 1962.
Knuth, op. cit. (General section).

番地計算法とハッシュ法

Flores, I.: "Computer Time for Address Calculation Sorting," *JACM*, 7, 1960, pp. 389-409.

Heising, W.P.: "Note on Random Addressing Techniques," *IBM Systems Journal*, vol. 2, June 1963, pp. 112-116.

Johnson, L.R.: "An Indirect Chaining Method for Addressing on Secondary Storage Keys," *CACM*, vol. 4, no. 5, May 1961, pp. 218-222.

Maurer, W. D.: "An Improved Hash Code for Scatter Storage," *CACM*, vol. 11, no. 1, Jan. 1968, pp. 35-38.

Morris, Robert: "Scatter Storage Techniques," *CACM*, vol. 11, no. 1, January 1968, pp. 38-44.

Peterson, W. W.: "Addressing for Random Access Storage," *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130-146, April 1957.

Schay, G. Jr. and W. G. Spruth: "Analysis of a File Addressing Method, *CACM*, vol. 5, no. 8, August 1962, pp. 459-462.

比較分類法

Grassner, Betty Jane: "Sorting by Replacement Selection," *CACM*, vol. 10, no. 2, February 1967.

Shell, D. L.: "A High Speed Sorting Procedure," *CACM*, vol. 2, no. 1, January 1959, pp. 30-32.

分配分類法

MacLaren, Donald M.: "Internal Sorting by Radix Plus Shifting," *JACM*, vol. 13, no. 3, July 1966, pp. 404-411.

2分探索法と樹状分類法

Arora, S. R. and W. T. Dent: "Randomized Binary Search Technique," *CACM*, vol. 12, no. 2, February 1969, pp. 77-80.

Clampett, H. A., Jr.: "Randomized Binary Searching with Tree Structures," *CACM*, vol. 7, no. 3, March 1964, pp. 163-165.

Patt, Yale N.: "Variable Length Tree Structures Having Minimum Average Search Time," *CACM*, vol. 12, no. 2, February 1969, pp. 72-76.

Sussenguth, E. H., Jr.: "Use of Tree Structures for Processing Files," *CACM*, vol. 6, no. 5, May 1963, pp. 272-279.

マクロ・プロセッサ

本書では伝統的なマクロ・プロセッサを示した．マクロ・プロセッサはテキスト編集プログラムとみなすこともできる．読者は各種のテキスト編集プログラム――“ライン・エディタ”，“ 文脈エディタ ”，“QED型エディタ”，TRAC など――を調べてみてほしい．

特に興味深いのは **Halpern** および **Kent** の論文である．**Wegner** の教科書の第２章および第３章はマクロを詳しく述べている．**Dijkstra** は回帰性を，**Rosenbloom** の本は数式論理に関する丁寧な註訳の付いた文献一覧表をのせている．

Brown, P. J.: "The ML/I Macro Processor," *CACM*, vol. 10, no. 10, 1967, pp. 618-623.

Dijkstra, E. W.: "Recursive Programming," in Rosen, Saul (ed): *Programming Systems and Languages*, McGraw-Hill, New York, 1967.

Farber, D. J.: R. E. Griswold, and I. P. Polonsky: "SNOBOL, A String Manipulation Language," *JACM*, vol. 11, no. 2, 1964, pp. 21-30.

Graham, M. L., and P. Z. Ingerman: "A Universal Assembly Mapping Language," *Proceedings, ACM 20th National Conference*, 1965, pp. 409-421.

Greenwald, I. D.: "A Technique for Handling Macro Instructions," *CACM*, vol. 2, no. 11, 1959, pp. 21-22.

Halparn. M.: "XPOP: A Meta Language Without Metaphysics," *Proceedings, AFIPS, 1964 FJCC*, pp. 57-68.

Kent, William: "Assembler—Language Macroprogramming," *Computing Surveys*, vol. 1, no. 4, 1969, pp. 183-196. (See also Dorn's "Editor's Preview" to the article.)

Maurer, Ward Douglas: "The Compiled Macro Assembler," *Proceedings, AFIPS, 1969 SJCC*, pp. 89-93.

McIlroy, M. D. : "Macro Instruction Extensions of Compiler Language," *CACM*, vol. 3, no. 4, 1960, pp. 214-220.

Mooers, C. N.: "TRAC, A Procedure Describing Language for the Reactive Typewriter," *CACM*, vol. 9, no. 3, 1966, pp. 215-219.

Rosenbloom, P.: *The Elements of Mathematical Logic*, Dover Publications, 1950.

Strachey, C. : "A General Purpose Macrogenerator," *Computer Journal*, vol. 8, no. 3, 1965, pp. 225-241.

ロ　ー　ダ

McCarthy, John, Fernando J. Corbató and Marjorie M. Daggett: "The Linking Segment Subprogram Language and Linking Loader," *CACM*, vol. 6, no. 7, June, 1963, pp. 391-395.
McGee, W. C.: "On Dynamic Program Relocation," *IBM Systems Journal*, vol. 4, no. 3, 1965, pp. 184-199.

プログラム言語

読者によっては本書に示した以外の言語の特徴を調べたいと思う人があるであろう．これは言語の比較を行ない，基本的な相違点に注意を向けるのに有用である．これらの一部を示しておく．LISP（リスト処理言語），SNOBOL（記号処理言語），AMBIT（2次元言語），BCPL（型のない言語），GPL（拡張言語），および APL（会話型言語）．

IBM System 360 PL/I Reference Manual, Form Mo. C28-8201-0.
Corbató, F. J.: "PL/I As a Tool for Systems Programming"; DATAMATION vol. 15, no. 5, May 1969, pp. 68-76.
Iverson, Kenneth E.: *A programming Language*, Wiley, 1962.
McCarthy, J. et al.: *LISP 1.5 Programmer's Manual*, M. I. T. Press, Cambridge, Mass., 1962.
Rosen, op. cit. (General section).
Sammet, Jean E.: *Programming Languages: History and Fundamentals*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

形式的体系とオートマトン

これはきわめて範囲の広い分野であるが，本書で焦点を合わせたのはほんの接点の部分にすぎない．Rosenbloom（マクロ・プロセッサの節を参照せよ）は論理回路の分野で丁寧に註釈を付けた文献をあげている．Curry と Feys は形式的体系に対する詳細な研究を述べている．本書ではオートマトンの理論については触れなかった．この分野の入門書としては Ginzburg または Minsky を参照してほしい．興味ある研究分野としては，解釈プログラムや言語の複雑さ，コンパイラの自動生成，非決定性の理論および計算機の構造の解析と設計をあげることができる．

Minsky と Papert の**パーセプトロン**を Newell が評論した本は一読の価値

がある．Newell は計算機科学の目的，研究法および方法論について論じている．

Birkhoff, G., and T. Bartee, *Modern Applied Algebra*, McGraw-Hill, New York, 1967.

Burstall, R. M.: "Proving Properties of Programs by Structural Induction", *Computer Journal*, vol. 12, no. 1, February 1969, pp. 41-48.

Curry, H. B., and R. Feys: *Combinatory Logic*, North Holland Publishing Co., Amsterdam, 1958.

Davis, Martin: *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

Floyd, Robert W.: "On the Non-existence of a Phrase-structure Grammer for ALGOL 60", *CACM* vol. 5, no. 9, 1962, pp. 483-484.

Ginsburg, Seymour: *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966.

Ginzburg, Abraham: *Algebraic Theory of Automata*, Academic Press, New York, 1968.

Hopcroft, John E., and Jeffrey D. Ullman: *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass., 1969.

Minsky, Marvin: *Computation — Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, N. J., 1967.

Minsky, Marvin, and Seymour Papert: *Perceptrons, An Introduction to Conputational Geometry*, M. I. T. Press, Cambridge, Mass., 1969.

Newell, Allen: "A Step Toward the Understanding of Information Processes", *Science*, 165:3895, p. 780.

Post, Emil L.: "Absolutely Unsolvable Problems and Relatively Undecidable Propositions — Account of an Anticipation", in Martin Davis, *The Undecidable*. Raven Press, Hewlett, New York, 1965.

Post, Emil L.: "Formal Reductions of the General Combinatorial Decision Problem", *American Journal of Mathematics*, 65, 1943, pp. 197-215.

Rogers, Hartley Jr.: *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

Smullyan, Raymond M.: "Theory of Formal Systems", *Annals of Mathematics Studies*, no. 47, Princeton University Press, Princeton, N. J., 1961.

正 準 体 系

現在のところ言語定義を目的としたこの方法の研究はまだ十分になされていない．Minsky は Post の体系に関連した議論を行ない，Smullyan は同様な体系の形式数学的な性質を述べている．形式的体系とオートマトンの節に示し

た Davis の本にのっている Post の論文をも参照せよ.

Alsop, Joseph W.: "A Canonic Translator", Project MAC Technical Report MAC-Tr-46, M. I. T., Cambridge, Mass., 1967.

Donovan, John J., and Henry F. Ledgard: "A Formal System for the Specification of Syntax and Translation of Computer Languages" *Proceedings, AFIPS* 1967 *FJCC*, pp. 553-569.

Haggerty, Joseph P.: *Complexity Measures for Language Recognition by Canonic Systems*, S. M. Thesis, M. I. T., Cambridge, Mass., 1969.

コンパイラ

ALGOL は多くの意味において **PL/I** の先駆者であり，初期の文献に多く取り上げられている．翻訳プログラム作成システムに関する研究の詳しい文献表は **Feldman** と **Gries** の本に示されている．コンパイラの研究には，基本として，本書のプログラム言語の節で示したモデルを用いるのもよい．また，このモデルの汎用性について考察してみるのもよいであろう．

最適化については機械に関係するものと機械に独立なものに関して幾つかの技法を述べた．別の最適化の方法は"発見的"といわれるものである．このような最適化を行なうコンパイラは自分で自分自身を適応させてその環境に合った最適化コードを生成する．読者はこの分野を研究目標として選ぶのもよい．

Arden, Bruce W., Bernard A. Galler, and Robert M. Graham: "The MAD Definition Facility" *CACM*, vol. 12, no. 8, August 2, 1969, pp. 432-439.

Bell, James R.: "A New Method for Determining Linear Precedence Functions for Precedence Grammars", *CACM*, vol. 12, no. 10, October, 1969, pp. 567-569.

Breuer, Melvin A.: "Generation of Optimal Code for Expressions Via Factorization", *CACM*, vol. 12, no. 6, June, 1969, pp. 333-340.

Dijkstra, E. W.: "Recursive Programming", in Rosen, *Programming Systems amd Languages*, McGraw-Hill, New York, 1967, pp. 221-227.

Feldman, J. and D. Gries, "Translator Writing Systems" *CACM*, vol. 11, no. 2, February, 1968.

Finkelstein, Mark: "A Compiler Optimization Technique", *Computer Journal*, vol. 2, no. 1, May 1968, pp. 22-25.

Freiburghouse, R. A. : "The MULTICS PL/I Compiler", *Proceedings, AFIPS, 1969, FJCC*, pp. 187-199.

Gear, C. W.: "High-speed Compilation of Efficient Object Code", *CACM*, vol. 8, no. 8, August 1965, pp. 483-488.

Gries, D., M. Paul and H.R. Wiehle: "Some Techniques Used in the Alcor Illinois 7090", *CACM*, vol. 8, no. 8, August 1965.
Gries, D., : *Compiler Construction for Digital Computers*, Wiley, New York, 1971.
　訳　書：牛島和夫訳・**コンパイラ作成の技法**（日本コンピュータ協会, 1976）
Hopgood, F.R.A.: *Compiling Techniques*, American Elsevier Publishing Co., Inc., New York, 1969.
Ingerman, Peter Z.: *A Syntax-Oriented Translator*, Academic Press, New York, 1966.
Jensen, Jorn: "Generation of Machine Code in ALGOL Compilers", *BIT*, vol. 5, no. 4, 1965, pp. 235-245.
Kanner, H., P. Kosinski, and S.L. Robinson: "The Structure of Yet Another ALGOL Compiler" *CACM*, vol. 8, no. 7, July 1965, p. 427.
Lowry, Edward S., and C.W. Medlock: "Object Code Optimization", *CACM*, vol. 12, no. 1, January 1969, pp. 13-22.
Nievergelt, J. : "On the Automatic Simplification of Computer Programs", *CACM*, vol. 8, no. 6, June 1965, pp. 366-370.
Perlis, Alan J. : "Procedural Languages" *Proceedings, 2nd Congress Information Systems Science*, pp. 189-210.
Randell, B. and L.J. Russell: *ALGOL 60 Implementation*, Academic Press, New York, 1964.
Randell, B., and L.J. Russell: "Single-Scan Techniques for the Translation of Arithmetic Expressions in ALGOL 60", *JACM*, vol. 11, no. 2, April 1964, pp. 159.

オペレーティング・システム

本節に示す参考文献はシステム設計の多くの面を取り扱っている．もちろんシステム設計と計算機の構造の考察には重複したところが存在する．CACM の1968年5月号（Vol. 11, No. 5）はオペレーティング・システムに全紙面を費している．入門のためには Wilkes および Needham (Computer Journal) の論文を参照せよ．また一般の節の Rosen をも参照せよ．

本書では，オペレーティング・システムを資源管理の面から述べた．研究の分野として，読者は計算機システムの4つの資源（記憶装置，プロセッサ，入出力装置，および情報）の管理に適用できる一般理論と数学を研究してみるのもよい．

Arden, B. W., B. A. Galler, T. C. O'Brien and F.H. Westervelt: "Programming and Addressing Structure in a Time-Sharing Environment" *JACM*, vol. 13, no. 1, January 1966, pp. 1-16.

Bright, Herbert S.: "A Philco Multiprocessing System" *Proceedings, AFIPS 1964 FJCC*, Part II, pp. 97-141.

Clayton, B. B., E. K. Dorff and R. E. Fagen: "An Operating System and Programming Systems for the 6600", *Proceedings, AFIPS 1964 FJCC*, Part II, pp. 41-57.

Coffman, E. G., Jr., and L. Kleinrock: "Computer Scheduling Methods and Their Countermeasures", *Proceedings, AFIPS 1968 SJCC*, pp. 11-21.

Coffman, E. G, Jr., and R.R. Muntz: "Models of Pure Time-Sharing Disciplines for Resource Allocation", *Proceedings, ACM 24th National Conference*, pp. 217-228.

Connors, Thomas L., and Donald E. Walker, (eds.): "Software Concerns in Advanced Information Systems", in *Information System Science and Technology*, Thompson Book Co., Washington, D. C., 1967, pp. 395-398.

Corbató, F. J. and V. A. Vyssotsky: "Introduction and Overview of the MULTICS System, *Proceedings, AFIPS 1965 FJCC*, pp. 185-196.

Daley, Robert C., and Jack B. Dennis: "Virtual Memory, Processes, and Sharing in MULTICS", *CACM*, vol, 11, no. 5, May 1968, pp. 306-312.

Denning, Peter J. : "Thrashing" Its Causes and Prevention", *Proceedings, AFIPS 1968 FJCC*, Part I, pp. 915-922.

Denning, Peter J. : "The Working Set Model for Program Behavior", *CACM*, vol, 11, no. 5, May 1968, pp. 323-333.

Dennis, Jack B. : "Segmentation and the Design of Multiprogrammed Computer Systems", *JACM*, vol. 12, no. 4, October 1965, pp. 589-602.

Dijkstra, Edsger W. : "The Structure Of the 'THE' Multiprogramming System", *CACM*, vol. 11, no. 5, May 1968, pp. 341-346.

Gibson, Charles T. : "Time-sharing in the IBM System/360 Model 67". *Proceedings, AFIPS, 1966 SJCC*, pp. 61-78.

Glaser, E. L. , J. F. Couleur and G. A. Oliver: "Systen Design of a Computer for Time-Sharing Applications", *Proceedings, AFIPS, 1968 FJCC*, Part I, pp. 197-202.

Graham, Robert M.: "Protection in an Information Processing Utility", *CACM*, vol. 11, no. 5, May 1968, pp. 365-369.

Hellerman, H. : "Some Principles of Time-sharing Scheduler Strategies", *IBM System Journal*, vol. 8, no. 2, 1969, pp. 94-117.

Kinslow, H. A. : "The Time-Sharing Monitor System", *Proceedings, AFIPS, 1964 FJCC*, pp. 443-454.

Kleinrock, L. : "Time-Shared Systems: A Theoretical Treatment", *Journal of the ACM*, vol. 14, no. 2, April 1967, pp. 242-261.

Lampson, Butler W. : "A Scheduling Philosophy for Multiprocessing Systems", *CACM*, vol. 11, no. 5, May 1968, pp. 347-360.

Madnick, Stuart, "Multiprocessor Software Lockout", *Proceedings, 1968 ACM* National Conference, pp. 19-24.

Mendelson, Myron J., and A. W. England: "The SDS SIGMA 7: A Real-Time Time-Sharing Computer", *Proceedings, AFIPS 1966 FJCC*, pp. 51-64.

Oppenheimer, G. and N. Weizer: "Resource Management for a Medium Scale Time-Sharing Operating System", *CACM*, vol. 11, no. 5, May 1968, pp. 313-322.

Randell, B. and C. J. Kuehner: "Dynamic Storage Allocation Systems", *CACM*, vol. 11, no. 5, May 1968, pp. 297-306.

Randell, B. : "A Note on Storage Fragmentation and Program Segmentation", *CACM*, vol. 12, no. 7, July 1969, pp. 365-369, 372.

Rosin, Robert F. : "Supervisory and Monitor Systems", *Computing Surveys*, vol. 1, no. 1, March 1969, pp. 37-54.

Sackman, H.: "Time-Sharing Versus Batch Processing: The Experimental Evidence", *Proceedings, AFIPS 1968 SJCC*, vol. 1, no. 10.

Saltzer, J. H. : "Traffic Control in a Multiplexed Computer System", MAC-TR-30 Thesis, Project MAC, Cambridge, Mass. July 1966.

Tonik, Albert B.: "Development of Executive Routines, Both Hardware and Software", *Proceedings, AFIPS, 1967 FJCC*, pp. 396-408.

Vyssotsky, V. A., F. J. Corbató and R. M. Graham: "Structure of the MULTICS Supervisor", *Proceedings, AFIPS, 1965 FJCC*, Part I, pp. 203-212.

Wichmann, B. A. : "A Modular Operating System", *Proceedings, IFIP Congress* 1968, Software 2, Booklet C. pp. 48-54.

Wilkes, M. V. : "The Design of Multiple-Access Computer systems (Part 1)", *Computer Journal*, vol. 10, no. 1, May 1967, pt. 1-9.

Wilkes, M. V. and R. M. Needham: "The Design of Multiple-Access Computer Systems (Part 2)", *Computer Journal*, vol. 10, no. 4, February 1958, pp. 315-320.

Wilkes, M. V. : "A Model for Core Space Allocatiom in a Time-Sharing System", *Proceedings, AFIPS, 1969, SJCC*, pp. 265-271.

Wilkes, M. V. : *Time-Sharing Computer Systems* American Elsevier Publishing Company, New York, 1968.

Witt, Bernard I. : "M65MP: An Experiment in OS/360 Multiprocessing", *Proceedings, ACM 23rd National Conference*, 1968, pp. 691-703.

Wood, Thomas C.: "A Generalized Supervisor for a Time-Shared Operating System", *Proceedings, AFIPS, 1967 FJCC*, pp. 209-214.

保護とプロセス間の通信

de Bruijn, N. G.: "Additional Comments on a Problem in Concurrent Programming Control", Letter to the Editor, *CACM*, vol. 10, no. 3, March 1967, pp. 137-138.

Corbató, F. J., and J. H. Saltzer: "Some Considerations of Supervisor Program Design for Multiplexed Computer Systems", *Proceedings, IFIP Congress* 1968, Invited Papers, pp. 66-71.

Dijkstra, E. W.: "Solution of a Problem in Concurrent Programming Control", *CACM*, vol. 8, no. 9, September 1969, p. 569.

Habermann, A. N.: "Prevention of System Deadlocks", *CACM*, vol. 12, no. 7, July 1969, pp. 373-377, 385.

Knuth, Donald E.: "Additional Comments on a Problem in Concurrent Programming Control", Letter to the Editor, *CACM*, vol. 9, no. 5, May 1966. pp. 321-322.

Lampson, B. W.: "Dynamic Protection Structures", *Proceedings, 1969 FJCC*, pp. 27-28.

Ossanna, J. F., L. E. Mikus and S. D. Dunten: "Communications and Input/Output Switching in a Multiplex Computing System", *Proceedings, AFIPS, 1965 FJCC*, Part 1, pp. 231-241.

Randell, B. and C. J. Kuehner: "Dynamic Storage Allocation Systems", *CACM*, vol. 11, no. 5, May 1965, pp. 297-306.

ファイル・システム

Chapin, Ned: "A Comparison of File Organization Techniques", *Proceedings, ACM 24th National Conference*, pp. 273-283.

Daley, R. C. and P. G. Neumann: "A General-Purpose File System for Secondary Storage" *Proceedings, 1965 FJCC*, Part 1, pp. 213-229.

Madnick, Stuart E. and Joseph W. Alsop: "A Modular Approach to File System Design", *Proceedings, AFIPS 1969 SJCC*, vol. 1, no. 13.

Madnick, Stuart E.: "Design Strategies for File System", MIT Project MAC report TR78, Cambridge, Mass., Oct. 1970.

Sundeen, Donald H.: "General Purpose Software" *Datamation*, vol. 14, no. 1, January 1968, pp. 22-27.

Weingarten, Allen: "The Analytical Design of Real-Time Disk Systems", *Proceedings, IFIP Congress* 1968, Hardware 1, Booklet D, pp. 131-137.

Wilkes, M. V.: "A Programmer's Utility Filing System", *Computer Journal*, vol. 7, no. 3, October 1964, pp. 180-184.

付　録

結合の規約

1.　レジスタの内容の退避と復元

IBM　360 において定められた規約によると，すべてのプログラムやサブルーチンがそれを呼出したプログラムに制御を返却するとき，汎用レジスタ 2 から 14 の内容は，そのルーチンに入ったときと同一でなければならない．したがって，1 つのルーチンに入ったら直ちに汎用レジスタに納められている値を退避し，制御を返却する前にこれらの値を復元しておくことが標準的なプログラムの習慣とされている．制御がオペレーティング・システムから問題プログラムへ，そしてサブプログラムへと渡されるとき，レジスタの内容は各段階で保存される．

次に示すレジスタには，プログラム結合を行なうために特別に割当てられた役割りがある：

レジスタ15――呼出されたプログラムの入口（復帰に際しては復帰コードを入れることができる）．

レジスタ14――呼出したプログラムの中の復帰場所．

レジスタ13――呼出されたプログラムが汎用レジスタを退避するための退避領域の番地．

レジスタ 0 および 1――引数を渡すために用いてもよい（レジスタ 1 は通常引数の番地の一覧表の先頭番地を納めている）．

2. 退 避 領 域

制御がサブプログラムに移されたとき，呼出したプログラムは規約によって，呼出されたプログラムが汎用レジスタを格納するための“退避領域”を用意しなければならない．レジスタ13には，“退避領域レジスタ”として，呼出し手順を実行する前に退避領域の番地を設定しておかなければならない．退避領域は制御が次々と下のレベルのプログラムに移されるにつれて連鎖される．このようにすると上のレベルのプログラムへ制御が復帰するとき，レジスタの内容を復元することができる．退避領域は18語を占め，語境界に整列し，**図 A.1** に示したような標準書式になっていなければならない．

語	変位	内容
1	0	PL/I プログラムだけで使用される標示子
2	4	呼出しプログラムの退避領域の番地（前）
3	8	呼出されたプログラムの退避領域の番地（次）
4	12	復帰番地（レジスタ14）
5	16	入口（レジスタ15）
6	20	レジスタ 0
7	24	レジスタ 1
8	28	レジスタ 2
9	32	レジスタ 3
10	36	レジスタ 4
11	40	レジスタ 5
12	44	レジスタ 6
13	48	レジスタ 7
14	52	レジスタ 8
15	56	レジスタ 9
16	60	レジスタ10
17	64	レジスタ11
18	68	レジスタ12

図 A・1　退避領域の最初の18語の内容

3. 標準システムの結合規約

プログラムへ入るときの結合は4つの機能を果さなければならない：

1. レジスタ14，15および0から12をレジスタ13で指された退避領域に

格納する.

2. ベース・レジスタの値を設定する.
3. 退避領域の番地をレジスタ13に設定する.
4. 退避領域を連鎖するために番地を格納する.

プログラム実行を完了したときには次のことを行なって復帰しなければならない：

1. 呼出しプログラムの退避領域の番地をレジスタ13に復元する.
2. レジスタ14，15および0から12を復元し，制御を復帰する.

次の一連のコードは，アセンブラ語のプログラムの入口と出口とにおける標準的な結合を示している：

```
EXAMPLE  START
         USING  *,15
         STM    14,12,12(13)   レジスタを退避
         LR     11,13          SAVEAREAの番地を複写
         LA     13,SAVEAREA    新しいSAVEAREAの番地を求める
         ST     13,8(0,11)     SAVEAREAの番地を呼出し
                               プログラム中に記入
         ST     11,4(0,13)     呼出しプログラムのSAVEAREA
                               の番地を現在のSAVEAREAに記入
         ⋮
         L      13,4(0,13)     元のSAVEAREAの番地を復元
         LM     14,12,12(13)   レジスタを復元
         BR     14             呼出し者に復帰
SAVEAREA DS     18F
         END
```

註：最もレベルの低いサブルーチン，すなわち，他のサブルーチンを呼出さないサブルーチンは新しい退避領域を作らないので，次のような簡略な手順を使用することができる：

```
EXAMPLE  START
         USING  *,15
         STM    14,12,12(13)   レジスタを退避
         ⋮
         LM     14,12,12(13)   レジスタを復元
         BR     14
         END
```

4. 例題プログラムの結合

サブルーチン結合の完全な過程を説明するために，次の簡単な FORTRAN サブルーチンを標準の結合規約を用いて対応するアセンブラ語に翻訳する：

```
//      EXEC     FORTRAN
            INTEGER    I,J,SUM
            DO     100        I = 1, 25
            DO     100        J = 1, 50
            CALL       ADD(SUM,I,J)
100         CONTINUE
            END

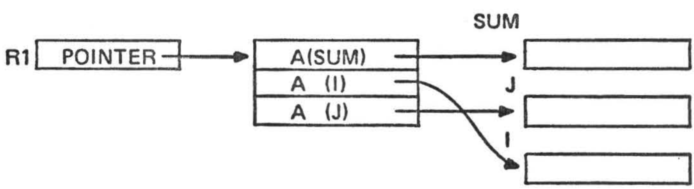
//      EXEC     FORTRAN
            SUBROUTINE  ADD (X,Y,Z)
            INTEGER        X,Y,Z
            X = Y + Z
            RETURN
            END
```

“MAIN” に等価なアセンブラ語プログラム

```
MAIN     START
         USING          *,15
         STM            14,12,12(13)
         LR             11,13
         LA             13,SAVE
         ST             13,8(0,11)
         ST             11,4(0,13)
         DROP           15
         BALR           10,0
         USING          *,10
IC       EQU            4
JC       EQU            5
IR       EQU            6
JR       EQU            7
         L              IC,=F'25'
         L              IR,=F'1'
ILOOP    L              JC,=F'50'
         L              JR,=F'1'
JLOOP    ST             IR,I
         ST             JR,J
```

```
        LA      1,ARGS
        L       15,=V(ADD)
        BALR    14,15
        A       JR,=F'1'
        BCT     JC,JLOOP
        A       IR,=F'1'
        BCT     IC,ILOOP
        L       13,4(0,13)
        LM      14,12,12(13)
        BR      14
SUM     DS      F
J       DS      F
I       DS      F
ARGS    DC      A(SUM,I,J)
SAVE    DS      18F
        END
```

ADD に入るとき，引数は次に示したようにして渡される：



“ADD” に等価なアセンブラ語プログラム

```
ADD     START
        STM     14,12,12(13)
        BALR    BASE,0
        USING   *,BASE
AC      EQU     8
ADDR    EQU     9
BASE    EQU     10
        L       ADDR,4(0,1)
        L       AC,0(0,ADDR)     AC   ←  I
        L       ADDR,8(0,1)
        A       AC,0(0,ADDR)     AC   ←  AC + J
        L       ADDR,0(0,1)
        ST      AC,0(0,ADDR)     SUM  ←  AC
        LM      14,12,12(13)
        BR      14
        END
```

プログラム MAINは サブルーチンと同じ規約を用いている．主プログラムは，実はオペレーティング・システムのサブルーチンである．

5.　引数処理の組込み

第8章で述べたように，多くの高水準言語ではサブルーチンを初期化する導入部が存在する．この導入部の役割りの1つは引渡された引数の処理である．特に，IBM 360 では上に示したように引数は特別のレジスタ（R1）によって渡される．このレジスタは番地表を指している．呼出されたプログラムは幾とおりかの方法でこの引数を参照することができる．この方法を簡単に述べる：

1.　呼出されたプログラムは，引数レジスタ（R1）に間接的に適当な引数の番地を求めることができる．この技法は導入部でオーバヘッドがないという利点がある．レジスタ1は必要なすべての情報を持っている．しかし次の2つの欠点がある：
    a.　ポインタ・レジスタ R1 を変更してはならない．
    b.　すべての参照は間接的に行なわなければならない．上のFORTRAN プログラムではこの方法を用いた（すなわち，L ADDR, 4(0, 1) として，次に L AC, 0(0, ADDR) を実行して I を求める)．
2.　導人部が引数表（番地の一覧表）を，呼出されたプログラムで使用されているスタックまたは記憶域の中に複写する．このようにするとポインタ・レジスタ R1 は解放される．しかし依然として間接的なアクセスを要する欠点が残る．導入部のオーバヘッドは小さい．この技法は IBM PL/I(F) 第5版のコンパイラで使用されている．
3.　導入部は引数の実際の値を呼出されたプログラムで使用する記憶域に複写することができる．この方法の利点はポインタ・レジスタが解放され，アクセスのために間接指定が不要であることである．しかしながら，導入部のオーバヘッドが大きいことと，引数の値を復帰に際して写し直さなければならないことが欠点である．この方法は**図 8.1** の MINI-PL/I のコンパイルで用いていた．

一般にプログラム言語は，引数の受渡しに次の3つの方法を規定することができる．これらには別々の組込みが必要となる．

1. “場所による呼出し”（“番地による呼出し”とも“参照による呼出し”とも呼ばれる）：言語では引渡されるものが引数の番地であることを規定している（例えば，上述のFORTRANの例題など）．この場合上述の1または2の技法を用いて組込むことができる．
2. “値による呼出し”：言語では引渡されるものが引数の値であること規定している．この場合上述の技法3を用いて組込むことができる．
3. “名前による呼出し”：言語では，ALGOL 60 で定めているように，

引数が使用される度にその値を計算し直すことを規定している．

なお，PL/I (F) 第5版のコンパイラは，第8章（図8.1）で用いた例題のMINI-PL/I プログラムをコンパイルするとき，誤り警告メッセージを出す．例の PROCEDURE 文の中で，RATE, START, FINISH はこのプログラムに渡される引数である．にも拘らず，次の文でこれらの変数に対してこのプログラムの中に記憶域を宣言している．

これは“場所による呼出し”方法と矛盾しているが，問題プログラムのコンパイルが容易であったのでこのようにしておいた．

演習問題解答

本書の演習問題の解は一義的に定まるものが多いが，設計に関する問題の解答は幾とおりかの解が存在する．本節に示す解答は１つの標準的な解である．

第１章　背　　景

1.　プロセッサと手続き

プロセッサとは命令を解釈し，実行するハードウェアの装置である．手続きは命令の集合である．

手続きとプログラム

手続きとプログラムの間には差がない．

プロセッサと I/O チャネル

I/O チャネルはプロセッサの一種であり，I/O 命令を解釈する．

多重処理と多重プログラミング

多重プログラミングにおいては，プロセッサの数には無関係に，幾つかのプログラムの実行が他のプログラムの完了を待たずに開始される．多重処理においては，１つの共通な主記憶装置の情報を同時に使用して，複数台のプロセッサが動作する．プロセッサの台数と主記憶中のプログラムの数とは無関係である．

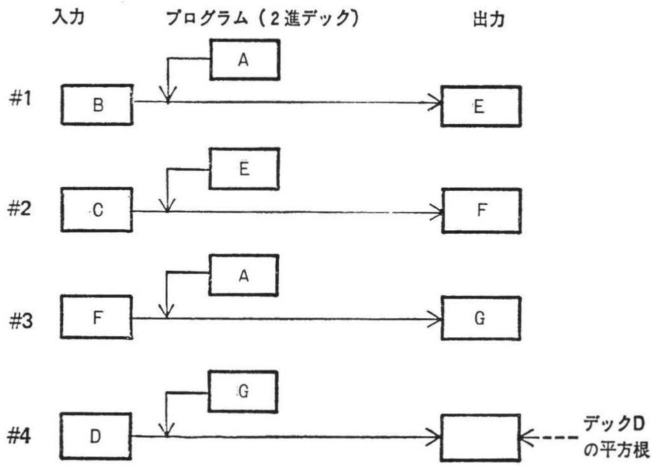
開いたサブルーチンと閉じたサブルーチン

開いたサブルーチンはアセンブル時またはコンパイル時にプログラムの中に挿入されるコードである．閉じたサブルーチンは別個のコードとして用意され，実行時に制御が渡される．

2.　ビットの集合はそれが使用されるときに初めてその意味が与えられる．プ

ログラマはビットの集合がいかに解釈されるべきかをプロセッサに通知する．プロセッサはロケーション・カウンタの内容にしたがって次の命令を取り出す．命令の番地部はデータを納めている番地を指す．同じ番地はあるときは命令として，別のときはデータとして解釈されることがある．

3.



4. ソフトウェア　a，c，d，g，h*，i
ハードウェア　b，e，f，j

＊ 時としてディスクなどのハードウェア装置を指すこともある．

5. a. 利用者には，システム・プログラムが作成されたとき考えられていたよりも広い範囲で装置を選択する機能が与えられる．また，システム構成の変更に際しても装置番号を合せるためにプログラムを書き直す必要がない．例えば，新しい印刷機が追加されたとすると1つの DD カードを変更するだけでこれを使用することができる．

b. データ・セット名とカードのパラメータの対応を付けるために1つの表が必要である．データ・セットの名前は幾つかのジョブ・ステップにおいて使用されることもある．

```
//SYSLIN DD FROM STEP1. SYSLIN
```

のようなカードを用いて名前が他のジョブ・ステップで参照されることがあるので，表はジョブ終了まで変更することができない．各名前にはジョブ・ステップ名を付けて格納しておく必要がある．表はジョブ終了時に消去される．

c. JOB カードに示された会計情報をジョブ終了時に消去されない表に保存しておかなければならない．EXEC カードのパラメータはそのジョブ・ステップの終了するまで保存しておかなければならない．

d. 制御が INIT/TERM に移されるのは：

i）DD ＊カードが出てきたとき（データ・カードは RDR/INT では読込まれず，実行されるプログラムで／＊が出てくるまで読込まれる）．

ii）次のジョブ・ステップの EXEC カードが出てきたとき（このカードの情報はそのジョブ・ステップが実行されるときのために保存しておかねばならない）．

iii）次のジョブの JOB カードが出てきたとき（このカードの情報は次のジョブが実行されるときのために保存しておかなければならない）．

e. 会計情報のデータ・ベースは保存される．

DD データ・ベースは保存される．

そのジョブ・ステップに関係する EXEC データ・ベースは消去される．

次のジョブやジョブ・ステップのデータ・ベースは保存される．

f. 会計情報のデータ・ベースは保存される．

DD データ・ベースは消去される．

EXEC データ・ベースは消去される．

g. RDR/INT：

カード書式の誤り（ジョブの排除）．

JOB カードに DD カードが続くようなカードの順番の誤り（ジョブの排除）．

存在しないプログラムやデータ・ベースの指定（ジョブの排除）．

INIT/TERM：

データ・セットの不十分な指定（可能ならば無指定の場合のデータ・セットを使用，またはジョブの排除）．

プログラムの続行不能な誤り（ジョブの排除）．

6. a. 各ジョブの PRIORITY は料金計算のために会計情報の表に保存しておかなければならない．

待ち行列はジョブに関連する DD カードや EXEC カードの情報を収容し，ジョブが終了すればそれを消去できるようにしておかねばならない．

b. この場合にはデータ・カードを待ち行列に読込み，保存しておかねばならない．その理由はプログラム実行中にデータ・カードを読込むことが物理的に不可能となるからである．

c. ３時間かかるジョブは，その実行のために高い料金を払うというのでなければ，短いジョブの実行を待たせてはならない．待つことによってプログラム実行の経費を軽減させることができ，料金を多く払うことによって速い応答を得ることができる．

第２章　計算機の構造，機械語およびアセンブラ語

1. a-1）OP1＝－26742

OP2＝26924

OP3＝－31891

OP2＞OP1＞OP3

a-2）OP1＝＋979

OP2＝＋692

OP3＝＋336

OP1＞OP2＞OP3

b-1）OP1＝20214

OP2＝24817

OP3＝－2574

OP2＞OP1＞OP3

b-2) OP1＝＋6

OP2＝－1

OP3＝12

OP3＞OP1＞OP2

2. 1) a. 0000 0101 0010 1100

b. (1024＋256＋32＋8＋4)＝1324

c. ＋52

d. BALR 2, 12

2) a. 0100 0101 0010 1100

b. (16384＋1024＋256＋32＋8＋4)＝17768

c. ＋452

d. 完全でない． X′45′ は BAL の命令コードであり，この命令の長さは 4 バイトである．

3) a. 0100 0101 0010 1000 0011 0110 0111 1101

b. (1, 393, 741, 824＋67, 108, 864＋16, 777, 216＋2, 097, 152
＋524, 288＋8192＋4096＋1024＋512＋64＋32＋16＋8＋4＋1)
＝1, 480, 263, 295

c. －4, 528, 367

d. BAL 2, 871(8, 3)

4) a. 0101 1001 0001 0100 1001 0111 0011 1100

b. (1, 393, 741, 824, ＋348, 435, 456＋174, 217, 728
＋16, 777, 216＋1, 048, 576＋262, 144＋32, 768
＋4096＋1024＋512＋256＋32＋16＋8＋4)
＝1, 934, 220, 800

c. ＋5, 914, 973

d. C 1, 1852(4, 9)

3. **a.** NEXT 部を取り除くためには，次に実行すべき命令の番地を入れる

ロケーション・カウンタを用いる．通常，次に実行すべき命令は記憶装置の中で次の番地に置かれている．したがって，1つの命令が実行されるとロケーション・カウンタは命令の長さだけ増加される．ループや条件判定のような番地の順でないプログラム実行を可能とするために，ロケーション・カウンタに新しい値を設定する分岐命令が必要である．

b.　RESULT 部を取り除くためには，結果を演算数1（または演算数2）に残せばよい．このようにすると演算数1は演算結果によって書き換えられてしまうので，元の値が後で必要となる場合にはコアに写しを保存しておかなければならない．このためにデータをコアに格納する命令が必要である．

c.　1つの演算数部は演算数の内の1つをアキュムレータに入れておくことによって取り除くことができる．このようにするとアキュムレータに1つの演算数を設定する命令が必要となる．

d.　ベース・レジスタとインデックス・レジスタがなく，番地指定のために24ビットが必要であるとする．1番地形式の命令は32ビットの長さとなり（命令コードに8ビット，番地に24ビット），一方，4番地形式命令は104ビットの長さとなる．1番地形式の命令の場合には（ロードや，格納や，分岐などの）追加の機能のために命令数が多くなるであろうが，それでもプログラムの長さは4番地形式の場合より短いと考えられる．

アキュムレータ（通常，高速の論理回路で構成され）を用いると何回もコアをアクセスする命令より実行が速い．しかし，アキュムレータの内容は頻繁に変更されるのでプログラマはその内容を保存しなければならない．

4.　a.　中間結果を1つのレジスタに残したまま他の演算を別のレジスタを用いて実行することができる．柔軟なインデックス・レジスタの指定が可能である．また，オフセットが4096バイト以上のときに幾つかのベース・レジスタを用いて番地指定が可能である．演算をレジスタ間で行なうことができる(これは主記憶をアクセスするより高速である)．

b. レジスタ0は，ベース・レジスタやインデックス・レジスタとして使用されるときにはレジスタが指定されなかったものとみなされる（すなわち，内容が0とされる）.

BCR や他の RR 形式の分岐命令でレジスタ0を指定すると分岐が起こらない.

演算命令はレジスタ0の内容を処理することができ，またレジスタ0に演算結果を格納することもできる.

5. **a.** USING はアセンブラに対する指令である．この命令でベース・レジスタ表への記入が行なわれ，この表を用いてアセンブラが記号番地からベース・レジスタとオフセットによる番地を作成する．USING に対して機械命令は生成されない.

BALR は実行される機械命令である．アセンブラは2バイトの機械命令を生成する．この命令は（名札が付けられているときにはこれが記号表に登録されることのほかは）アセンブラに何も情報を与えない.

b.

0	BALR	15, 0	0	BALR	15, 0
2	LR	10, 15	2	LA	10, 16(0, 15)
4	LH	1, 8(0, 10)	6	LH	1, 0 (0, 10)
8	BCR	15, 14	12	X′0000 0001′	
10	X′0001′		16	X′0000 0002′	
12	X′0002′		20	X′0000 0003′	
14	X′0003′				

REG1 は X′0000 0001′ を納めている.

アセンブラは，レジスタ10の内容が相対値4であると教えられている．しかし，レジスタ10は実行時には相対値2を

REG1 は X′0000 0002′ を納めている.

アセンブラはレジスタ15の内容が0であると教えられているが実際には2が設定されるので，DATA 2の番地が間違って（16であるべ

納めている．

きが18として）レジスタ内にロードされる．ベース・レジスタ10を用いる LH 命令は DATA 2 の左半語をロードする．

6. a. 働かない．乗算命令の第1演算数は偶数番号のレジスタでなければならない．

 b. 正しいこともある．被乗数はレジスタ2と3の中の64ビットの固定小数点数である．もしレジスタ2の内容が0であれば結果は正しい．

7. a. INDEX EQU 5はプログラムの中で INDEX が出てくるとアセンブラに値′5′を代入させるための擬似命令である．INDEX DC F′5′は，アセンブラが INDEX という名前の場所に5を出力するよう指示する擬似命令である．この値5は実行時にその場所に入っている．一方 EQU 5は実行時には存在しない．

 b. CR は2つの演算数を32ビットの符号付き固定小数点数として比較する．CLR は2つの演算数をビットごとに左から右への順で比較する．

8. a.

```
        LA    3,=A(XYZ)      XYZの番地を格納している場所の番地．
                             すなわち，リテラルA(XYZ)の番地
        LR    3,3            変更なし
        L     3,=F'5'        X'0000  0005'
XYZ     LCR   3,3            X'FFFF  FFFB'
        LNR   3,3            変更なし
```

 b. 次は LR 3,5 である．CLI 命令は第1演算数で指定された番地の1バイトを1バイトの直接演算数と比較する．リテラル＝F′3′は次の内容を生成する：

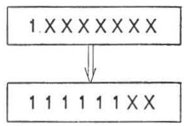
0000 0000	0000 0000	0000 0000	0000 0011
0　　7	8　　15	16　　23	24　　31

このデータの第1バイトと直接演算数

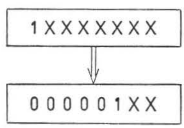
0000 0011
0　　7

が比較されると明らかに一致はとれない.

9. a. レジスタ0の最高位のビットが1のとき SRDA 0,5 はこれを符号ビットとして扱い次のようにシフトする：



SRDL 0,5 はシフトの後左に0を入れる：



b. SLL 1,1 はレジスタの内容を1ビット左へシフトする.
LA 1,0(1,1)はレジスタの内容を自分自身に加え，次に0を加える.
したがって，これらは上位の9ビットが0でなければ異なった結果を生じる.

c. MVC などの SS 形式の命令では，長さの部分は0（B′0000′）から15（B′1111′）の範囲の値をとり，1バイトから16バイトまでの長さを指定する. アセンブラは長さの指定が0以外のときは指定された長さから1を減じる. したがって，両方の命令は長さが B ′0000′ とされ1バイトの文字を移動させる.

d. LA 1,0(1) はレジスタ1の上位8ビットを0にする.
SRA 1,0 は条件コードを設定する.

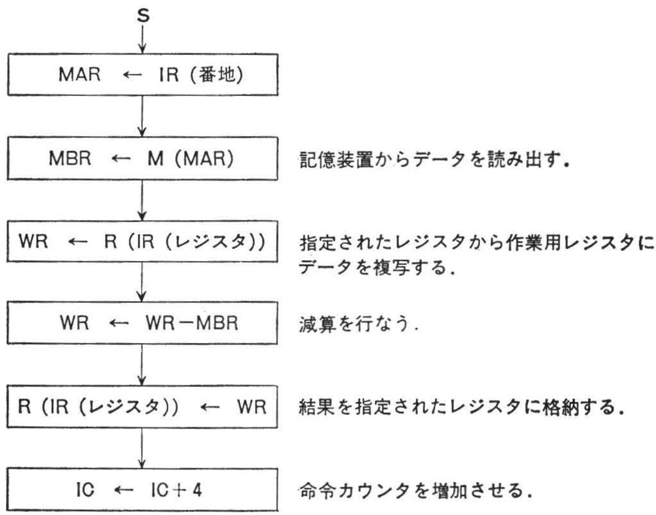
e. MVC は一時に1バイトずつ，左から右への順で，文字を第2演算数の位置から第1演算数の位置へ移動させる.

```
MVC    STOMP+1(8),STOMP    C'b̸b̸b̸b̸b̸b̸b̸b̸b̸'
MVC    STOMP(8),STOMP+1    C'ERASUREb̸b̸'
```

10. しない. 条件の設定や検査には時間がかかるので，たとえば，ロード命令は条件コードの設定も検査も行なわない. 条件コードは（たとえば，減算や，I/O 検査，判定などの）演算処理の結果の記録であり，後刻その結果

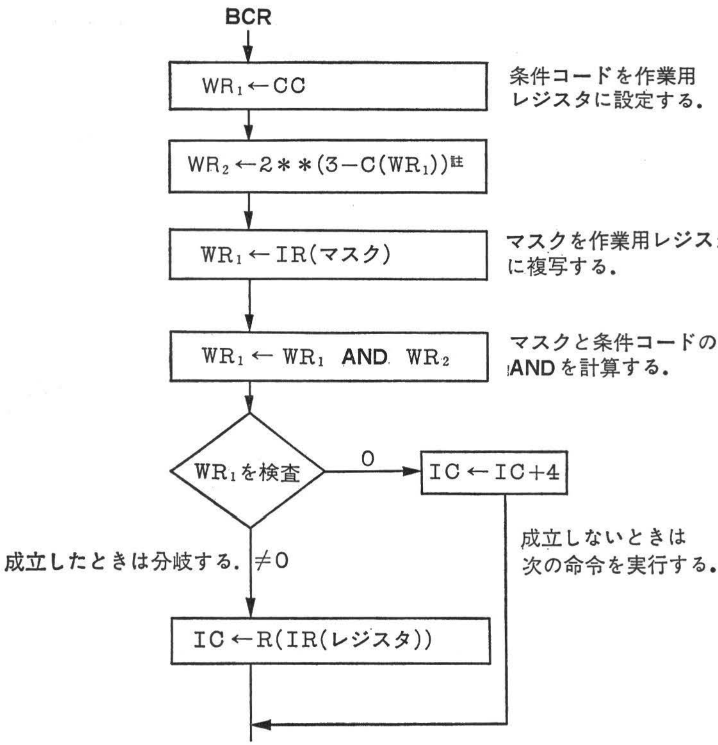
にしたがって条件分岐を行なうために使用される．条件コードには数の状態（＋，－，０），論理演算の結果，I/O 処理の状態などの情報を格納することができる．

11. a.　S（減算，RX 形式）



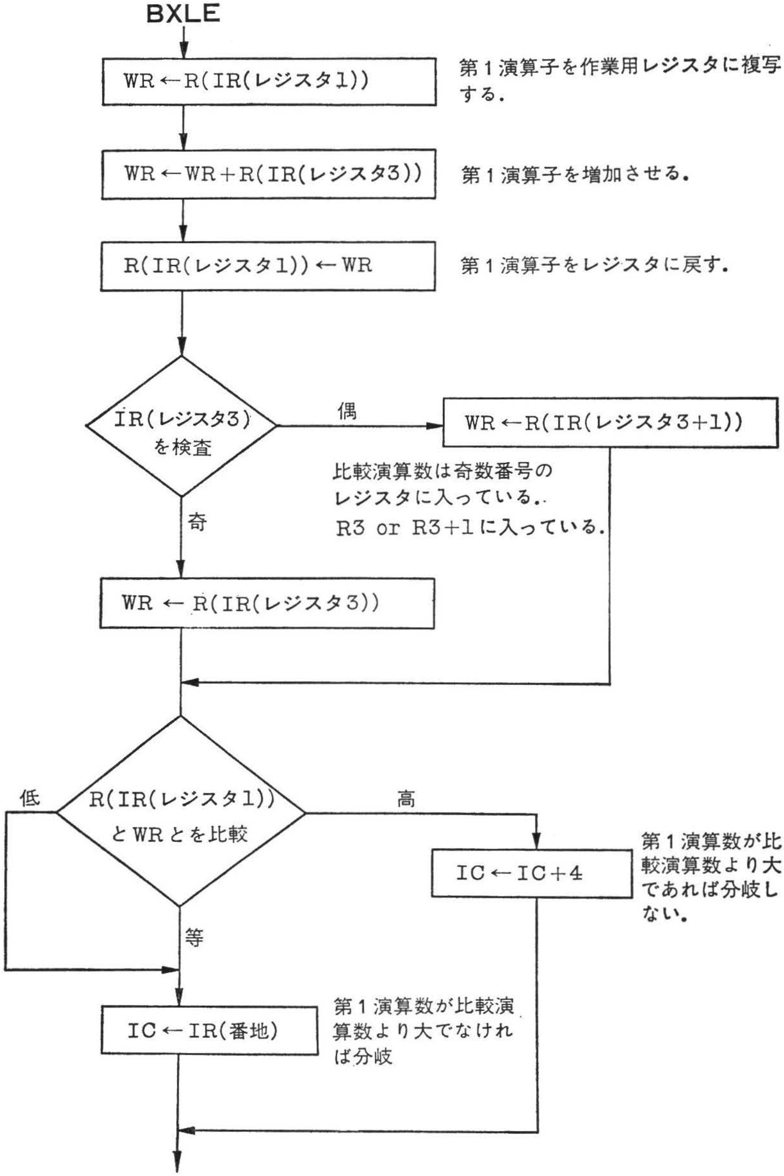
註：ここに示すマイクロ流れ図は IBM の計算機の処理をそのまま示しているわけではない．しかし，処理結果は計算機の処理結果と等しく，これらの命令がいかに処理されるかを説明している．

b.　BCR（条件分岐，RR 形式）

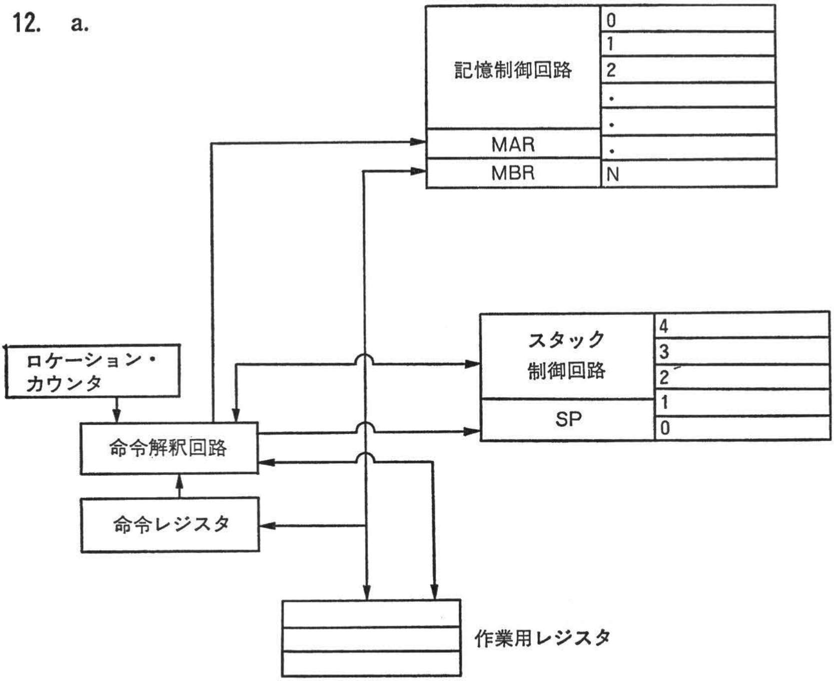


註：WR_2の右端から（4−C（WR_1））桁目にビット1を入れる.

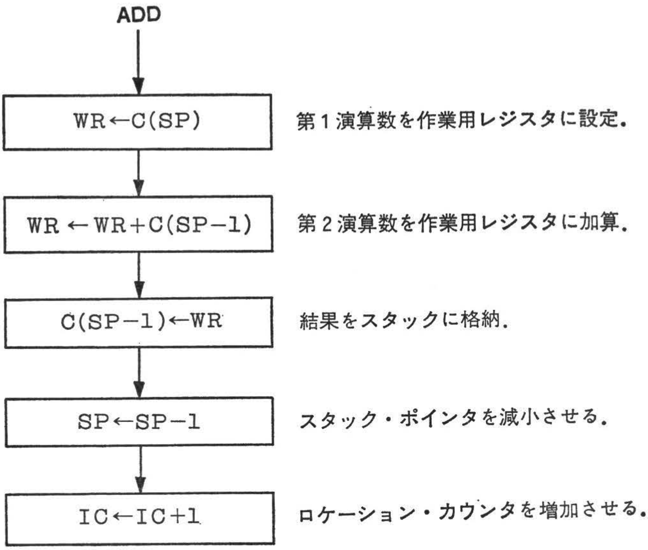
c.　BXLE（インデックス低等による分岐，RS 形式）



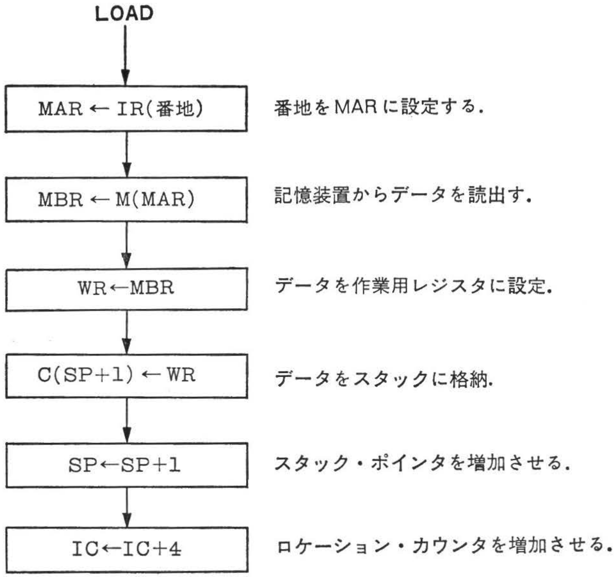
12. a.


記憶制御回路
MAR
MBR
0
1
2
.
.
.
N
スタック
制御回路
SP
4
3
2
1
0
ロケーション・
カウンタ
命令解釈回路
命令レジスタ
作業用レジスタ

b.


ADD
WR←C(SP)
第1演算数を作業用レジスタに設定.
WR ← WR+C(SP−1)
第2演算数を作業用レジスタに加算.
C(SP−1)←WR
結果をスタックに格納.
SP←SP−1
スタック・ポインタを減小させる.
IC←IC+1
ロケーション・カウンタを増加させる.

c.



13. a.

命　令	レジスタ1	レジスタ2	番地A
3	5	?	0
4	10	?	0
5	10	7	0
6	10	14	0
7	24	14	0
8	23	14	0
9	23	14	23

b.

```
3)   LM   1,2,B
4)   AR   1,2
5)   AR   1,1
6)   SH   1,=H'1'
7)   ST   1,A
```

8)　　…　　…

9)　　…　　…

註：半語のリテラルを使用していることに注意せよ.

14. 例えば UNIVAC 1108 の場合には次のようになる：

(1) **記憶装置**

基本単位	36ビット（パリティ 2ビット）
容　量	65k～256k
他の単位	(6, 9, 12, 18, 36, 72ビット)

(2) **レジスタ**

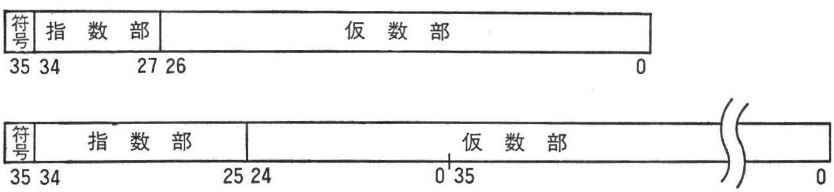
インデックス	15	共通レジスタ4
アキュムレータ	16	
特　殊	16	（時計，カウンタなど）

(3) **データ形式**

固定小数点：　1の補数

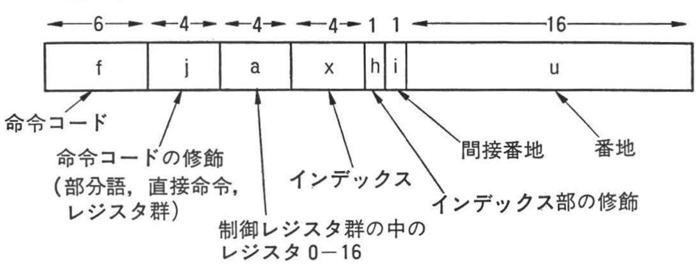
36, 72ビットほか

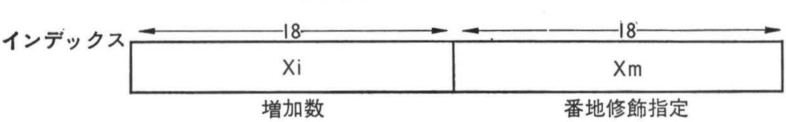
浮動小数点：



部分整数：　半語ほか

(4) **命　令**





実効番地＝(M＋C(X)m)＊　間接

H＝1 のときは命令コード解釈後に

$X_m \leftarrow X_m + X_i$

15. （採点プログラムを使用すると仮定する．）

STUDENT	START	0	
	ENTRY	STUDENTN	
	USING	*, 15	
	STM	2, 3, DATA	R2 および R3 をデータ領域に格納する．
	SR	1, 1	
	LA	9, DATA	これらの命令は BXLE 命令のための初期化を行なう．
	L	10, =F'1'	
	L	11, =F'7'	
LOOP	CLI	0(9), C','	
	BNE	CHECK	
	A	1, =F'1'	判定結果が真ならば R1 を増加させる．
CHECK	BXLE	9, 10, LOOP	
	BR	14	
DATA	DS	2F	R2 および R3 の内容をここに格納する．
	LTORG		
STUDENTN	EQU	*	
	END		

第3章　アセンブラ

1. 基本的には，記号が定義される前に参照されることがあるから（forward reference：前方参照）．たとえば，プログラムの後の方で出てくるラベルへの分岐や，プログラムの終りに置かれている EQU 文や DC 文で定義される記号の参照など．

2. 追加のパスがあると EQU 文の演算数においてまだ定義されていない記号を参照することができるようになる.

3パス・アセンブラの他の機能はコードの最適化ができることである. 追加したパスでは, 2を掛けるようなコードを調べてシフトに置換えたり, 次に示すように, 結果をレジスタに残しておく方が良いときには, 不必要に使用された格納命令を取り除く:

L	1, ALL		L	1, ALL
A	1, TEN		A	1, TEN
ST	1, TEMP			
L	1, JOHN	余分のパス→	L	2, JOHN
S	1, FOUR		S	2, FOUR
A	1, TEMP		AR	1, 2

上で変更されたプログラムは1命令短くなり, A命令はより高速の AR 命令に置換えられている. 第8章の最適化技法が適用できる.

追加したパスが最初のパスのときはある種の条件を検査することができる. 条件が満足されないとき, アセンブラはプログラムの一部分をアセンブルしない. たとえば, プログラマは広範囲のデータのためにプログラムを書いても, 狭い範囲のデータのためにプログラムをアセンブルしたいことがある. このときには原始プログラムの中にパラメータを設定しておき, アセンブラがそれを検査して必要な部分のみをアセンブルし, 能率の良いプログラムを作成することができる.

他の機能はマクロに関するものである (第4章参照).

最後に, 余分のパスは余計に計算機時間を消費することを付け加えておく.

3. a. 1

b. 1, 2

c. 1

d. 1, 2

e.　1

f.　2

g.　1, 2

h.　1, 2

i.　1, 2

j.　2

k.　2

4. a.　可.

b.　可.

c.　可. しかし記号は使用する前に EQU により定義しておかねばならない. また演算数部の記号は参照される前に定義しておかなければならない.

d.　c. に同じ

e.　可.

f.　可. ローダの責任である.

g.　可.

h.　不可（ローディングの段階で記号処理を行なうような特別のローダが用意されぬ限りは不可).

5. a.　書ける. すべてのベース・レジスタとオフセットとを規定しなければならない. 記号による参照はできない. これは記号命令名と10進の演算数を用いている以外は機械語のプログラミングを行なっていることに相当する.

b.　ロケーション・カウンタを次の2語境界に整合させる.

6. パス1において, 出てきたリテラルをリテラル表に登録する. LTORG が出てくると, これまでに集められたリテラルを処理し, 番地を割当てる . これを行なうためにリテラル表に1ビットのフラグ欄を設ける. リテラルをリテラル表に入れるときこのフラグを設定しておく. LTORG が出てくると未処理のリテラルを処理する. リテラル表はパス1にて作成しなければならない. その理由はロケーション・カウンタを正しく増加させておかねばならないからである.

パス2ではリテラル表のリテラルに対して機械コードを生成する.

LTORG の処理はパス1にて完了する.

7. a. パス1ではコードの生成が行なわれないが，ロケーション・カウンタの値を正しく増加させる情報をアセンブラに与える必要がある．したがって，DC 擬似命令の演算数の長さはパス1でこの命令を処理するときに必要となり，演算数に用いられるすべての記号は使用するより前に定義されていなければならない（または自己定義でなければならない).

b. EQU のラベル部の記号はパス2の開始時に値が定まっていなければならない．したがって，この値はパス1で EQU が出てきたときに割当てておかねばならない.

c. パス2になるまでは機械コードは生成されない.

定数を表わすビットと命令を表わすビットには本質的な相違がない.

8. 命令コードを機械命令表に追加しなければならない．そしてパス1でロケーション・カウンタの値を増加させるために命令の長さの情報が必要である.

パス2に対する機械命令表は，8ビットの命令コードだけでなく，4ビットのマスクの値と拡張記号であることを示す情報とが必要である.

別の方法として，拡張記号を擬似命令表に入れておきパス1で等価なマスク付きの分岐命令を中間テープに生成することもできる．次に，パス2は普通に処理を行なう．この方法は処理手順が増すが，機械命令の意味を変更しないのでわかりやすい.

9.　a.

記　号　表

	値	長さ	絶対／相対フラグ
SIMPLE	0	1	R
LOOP	2	4	R
R/	1	1	A
TWO	24	4	R
FOUR	28	4	R

リテラル表

空

ベ　ー　ス　表

レジスタ番号	内　容
15	2

機械コード

0	BALR	15,0
2	L	1,22(0,15)
6	A	1,22(0,15)
10	ST	1,26(0,15)
14	CLI	29(15),4
18	BC	7,0(0,15)
22	BCR	15,14
24	X'0002'	
28-31	確保した記憶域	

b.

記 号 表

	値	長さ	絶対／相対フラグ
SAE	488	1	R
ARCHON	1	1	A
DEPUTY	2	1	A
TREAS	3	1	A
BACK	EXTRN		
BEACON	494	2	R
BETA	548	6	R
STOMP	5	1	A
HOLE	592	8	R
POINT	592	4	R
FOOTBL	EXTRN		

リテラル表

	値	長さ	絶対／相対フラグ
A(BACK)	576	4	R
H'43'	588	2	R
A(BEACON)	580	4	R
F'482'	584	4	R
F'482'	640	4	R

(ここでLTORGが現われてリテラルが作成された)

ベース・レジスタ表

	レジスタ番号	内容
文6の後	3	486
文8の後	3	486
	1	548
	2	4644
文22の後	3	486
	2	4644

アセンブルされたコード

488	BALR	2, 0
490	LM	1, 6, 106(3)
494	CR	2, 1
496	BC	13, 38(0, 1)
500	LA	7, 28(0, 1)
504	CLI	44(1), X′90′
508	BCR	15, 6
510	X′0040′	
512	X′40′	
513	X′40	
514	X′40	
515	X′0000 0000 0000 0000′	
523	整合条件により飛ばされる	
524	X′0000 0186′	
528	X′0000 0186′	
532	X′0000 0186′	
536	X′0000 0186′	

540	X′0000 0186′	
544	X′0000 0186′	
548	MVC	64(4, 1), 40(1)
554	MVC	60(4, 1), 40(1)
560	L	9, 94(0, 3)
564	NR	9, 3
566	ST	9, 98(0, 3)
570	BC	15, 130(0, 3)
574	整合条件により飛ばされる.	
576	A(BACK)	
580	A(BEACON)	
584	X′0000 0188′	
588	X′001D′	
590	整合条件により飛ばされる.	
592	X′E6C9 D540′	
596	A(BETA)	
600	A(BETA)	
604	A(BETA)	
608	A(BETA)	
612	A(BETA)	
616	V(FOOTBL)	
620	CLC	102(4, 3), 154(3)
626	LA	1, 1(0, 0)
630	CVB	2, 10(6, 0)
634	STC	3, 106(9, 3)
638	BCR	15, 14
640	X′0000 0188′	

10. a. 両方のパスが変更される．記号表も多少変更され，DSECT 記号を処理するために番地形成手順を変更しなければならない．

b. DSECT 文が出てくるとロケーション・カウンタは 0 に設定され，

DSECT ラベルはフラグを付けて記号表に登録される．それに続くラベルは（DSECT ラベル以外は）フラグを付けずに登録される．

パス2では，USING 文が出てくると番地部の記号が DSECT ラベルであるか否かを調べる．もし DSECT ラベルであれば，記号表の中の，次の DESECT ラベルまでの，記号にフラグを立てる．そして，これらの記号の欄にベース・レジスタ番号を記入する．しかしベース・レジスタ表への記入は行なわない．記号参照を処理して番地を形成するとき，記号にフラグが立っていれば記号表に記入されたベース・レジスタ番号を指定する．記号にフラグが立っていなければ，ベース・レジスタ表からベース・レジスタが選定される．

c. 記号表に新しく2つの欄が必要である．
"DSECT フラグ" と "DSECT ベース・レジスタ"

11. 自分で解答してみよ．

12. a. 1) 探索法とは表から所要のデータを探し求めることである．
2) 分類法とは探索が容易なように表の内容を順に並べることである．
3) ハッシュ法とは分類法や探索法が必要でないような表の管理／参照の方法である．番地は見出し語から計算されるので，登録と参照の双方に用いることができる．

b. 分類は表の並べ換えである．探索はデータを探し求めることである．分類は探索の技法に含められる．たとえば，2分探索法のためには表は分類して並べておかねばならない．

13. (1) a.

81	52	52	22	22	04	04	04	04	04	04	04	04	04	04	04	04
52	~~81~~ 57	~~57~~ 22	52	~~52~~ 04	22	22	22	22	22	22	22	22	~~22~~ 16	16	14	14
57	~~81~~ 22	57	~~57~~ 04	52	52	42	~~42~~ 32	32	32	32	32	~~32~~ 16	22	~~22~~ 14	16	16
22	81	~~81~~ 04	57	57	57 42	~~52~~ 32	42	42	42	42	~~42~~ 16	32	~~32~~ 14	22	22	22
95	~~95~~ 04	81	81	~~81~~ 42	~~57~~ 32	~~52~~ 48	48	48	48	~~48~~ 16	42	~~42~~ 14	32	32	32	32
04	~~95~~ 83	83	~~83~~ 42	~~81~~ 32	~~57~~ 48	52	52	52	~~52~~ 16	48	~~48~~ 14	42	42	42	42	42
83	95	~~95~~ 42	~~83~~ 32	~~81~~ 48	57	57	57	~~57~~ 16	52	~~52~~ 14	48	48	48	48	48	48
96	~~96~~ 42	~~95~~ 32	~~83~~ 48	~~81~~ 78	78	~~78~~ 65	~~65~~ 16	57	~~57~~ 14	52	52	52	52	52	52	52
42	~~96~~ 32	~~95~~ 48	~~83~~ 78	81	~~81~~ 65	~~78~~ 16	65	~~65~~ 14	57	57	57	57	57	57	57	57
32	~~96~~ 48	~~95~~ 78	~~83~~ 82	~~82~~ 65	~~81~~ 16	~~78~~ 66	~~66~~ 14	65	65	65	65	65	65	65	65	65
48	~~96~~ 78	~~95~~ 82	~~83~~ 65	~~82~~ 16	~~81~~ 66	~~78~~ 14	66	66	66	66	66	66	66	66	66	66
78	~~96~~ 82	~~95~~ 65	~~83~~ 16	~~82~~ 66	~~81~~ 14	~~78~~ 77	77	~~77~~ 67	67	67	67	67	67	67	67	67
82	~~96~~ 65	~~95~~ 16	~~83~~ 66	~~82~~ 14	~~81~~ 77	78	~~78~~ 67	77	77	77	77	77	77	77	77	77
65	~~96~~ 16	~~95~~ 66	~~83~~ 14	~~82~~ 77	81	~~81~~ 67	78	78	78	78	78	78	78	78	78	78
16	~~96~~ 66	~~95~~ 14	~~83~~ 77	82	~~82~~ 67	81	81	81	81	81	81	81	81	81	81	81
66	~~96~~ 14	~~95~~ 77	83	~~83~~ 67	82	82	82	82	82	82	82	82	82	82	82	82
14	~~96~~ 77	~~95~~ 87	~~87~~ 67	83	83	83	83	83	83	83	83	83	83	83	83	83
77	~~96~~ 87	~~95~~ 67	87	87	87	87	87	87	87	87	87	87	87	87	87	87
87	~~96~~ 67	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95
67	96	96	96	96	96	96	96	96	96	96	96	96	96	96	96	96
	パス 1	パス 2	パス 3	パス 4	パス 5	パス 6	パス 7	パス 8	パス 9	パス 10	パス 11	パス 12	パス 13	パス 14	パス 15	パス 16 変更なし－終了

(2) a.

81	*48	*04	04	04	04
52	*52	*14	14	14	14
57	57	57	**~~48~~ 42	*~~42~~ 16	16
22	22	22	22	22	22
95	*16	16	16	*~~42~~ 32	32
04	04	*48	**~~57~~ ~~42~~ 48	48	*~~48~~ 42
83	*14	*52	*~~52~~ 32	*42	*48
96	*77	77	**~~77~~ ~~66~~ 65	*~~65~~ 52	52
42	42	42	*57	57	57
32	32	32	*52	*65	65
48	*81	*~~81~~ 66	**77 ~~65~~ 66	66	66
78	78	78	*~~78~~ 67	67	67
82	82	82	*~~82~~ 81	*~~81~~ 78	*~~78~~ 77
65	65	65	*77	77	*78
16	*95	*~~95~~ 67	*78	*81	81
66	66	*~~66~~ 81	*82	82	82
14	*83	83	83	83	83
77	*96	96	96	*~~96~~ 95	*~~75~~ 87
87	87	87	87	87	95
67	67	*95	95	*96	96
	パス 1 d=10	パス 2 d=5	パス 3 d=3	パス 4 d=2	パス 5 d=1

(3) a.

	第1回の分配	合併	第2回の分配	合併
81	0)	81	0) 04	04
52	1) 81	52	1) 14,16	14
57	2) 52,22,42,32,82	22	2) 22	16
22	3) 83	42	3) 32	22
95	4) 04,14	32	4) 42,48	32
04	5) 95,65	82	5) 52,57	42
83	6) 96,16,66	83	6) 65,66,67	48
96	7) 57,77,87,67	04	7) 77,78	52
42	8) 48,78	14	8) 81,82,83,87	57
32	9)	95	9) 95,96	65
48		65		66
78		96		67
82		16		77
65		66		78
16		57		81
66		77		82
14		87		83
77		67		87
87		48		95
67		78		96

(1) b.

424	424	424	424	424	424	424	~~424~~ 413	413	180	180	180
887	~~887~~ 807	~~807~~ 709	709	~~709~~ 616	~~616~~ 573	~~573~~ 413	424	~~424~~ 180	413	264	264
807	~~887~~ 709	807	~~807~~ 016	~~709~~ 573	~~616~~ 413	573	~~573~~ 180	424	~~424~~ 264	413	413
709	~~887~~ 882	~~882~~ 616	~~807~~ 573	~~709~~ 413	616	~~616~~ 180	573	~~573~~ 264	424	424	424
882	~~887~~ 616	~~882~~ 573	~~807~~ 413	~~709~~ 679	~~679~~ 180	616	~~616~~ 264	573	573	573	573
616	~~887~~ 573	~~882~~ 413	~~807~~ 679	~~709~~ 180	679	~~679~~ 264	616	616	616	616	616
573	~~887~~ 413	~~882~~ 679	~~807~~ 180	709	~~709~~ 264	679	679	679	679	679	679
413	~~887~~ 679	~~882~~ 180	807	~~807~~ 264	709	709	709	709	709	709	709
679	~~887~~ 180	882	~~882~~ 264	807	807	807	807	807	807	807	807
180	887	~~887~~ 264	882	882	882	882	882	882	882	882	882
975	~~975~~ 264	887	887	887	887	887	887	887	887	887	887
264	975	975	975	975	975	975	975	975	975	975	975
	パス 1	パス 2	パス 3	パス 4	パス 5	パス 6	パス 7	パス 8	パス 9	パス 10	パス 11

変更なし一終了

(2) b.

424	424	*180	180	180
887	*413	413	413	*~~413~~ 264
807	*679	*264	264	*413
709	*180	*424	424	424
882	882	882	*~~882~~ 573	573
616	*264	**~~679~~ 616	616	616
573	573	573	**~~882~~ 679	679
413	*887	887	*~~887~~ 709	709
679	*807	*~~807~~ ~~616~~ 679	*882	*~~882~~ 807
180	*709	709	*~~887~~ 807	882
975	975	975	975	*~~975~~ 887
264	*616	*807	*887	*975
	パス 1 d=6	パス 2 d=3	パス 3 d=2	パス 4 d=1

(3) b.

	第1回の分配	合　併	第2回の分配	合　併	第3回の分配	合　併
424	0) 180	180	0) 807,709	807	0)	180
887	1)	882	1) 413,616	709	1) 180	264
807	2) 882	573	2) 424	413	2) 264	413
709	3) 573,413	413	3)	616	3)	424
882	4) 424,264	424	4)	424	4) 413,424	573
616	5) 975	264	5)	264	5) 573	616
573	6) 616	975	6) 264	573	6) 616,679	679
413	7) 887,807	616	7) 573,975,679	975	7) 709	709
679	8)	887	8) 180,882,887	679	8) 807,882,887	807
180	9) 709,679	807	9)	180	9) 975	882
975		709		882		887
264		679		887		975

15. a.

	パス 1	パス 2	パス 3	パス 4	パス 5
00100 T1	00100 T2	00100 T3	*00010 T4	*00001 T5	*00000 T6,B6
10001	*01111	*00000	00000	00000 B5	*00001
01011	01011	*00101	*00001 B4	*00010	
00001	00001	00001	*00101		
00010	00010	00010 B3	*00100		
00101	00101	*01011			
00000	00000	*01111			
01001	01001 B2	01001			
10101	10101				
10010	10010				
01111	*10001				
11011 B1	11011				

第 1 分割点を示す.

＊は交換が行なわれたことを示す.

	パス 1	パス 2	パス. 3	パス 4	パス 5
00100 T1	00100 T2	00100 T3	*00010 T4	*00001 T5	*00000 T6,B6
10001	*01111	*00000	00000	00000 B5	*00001 T7,B7
01011	01001	*00101	*00001 B4	*00010 T8,B8	
00001	00001	00001	*00101 T9	00101 T10	*00100 T11,B11
00010	00010	00010 B3	*00100 B9	00100 B10	*00101 T12,B12
00101	00101	*01011 T9	01011 T13	*01001 T15,B15	
00000	00000	*01111	*01001 T14	*01011 T16,B16	
01001	01001 B2	01001 B9	*01111 T17,B17		
10101	10101 T18	10101 T19	*10001 T20	10001 T21,B21	
10010	10010	10010	10010 B20	10010 T22,B22	
01111	*10001	10001 B19	*10101 T23, B23		
11011 B1	11011 B18	11011 T24,B24			

最終的に並べ換えられた表

b.

	パス 1	パス 2	パス 3	パス 4	パス 5	パス 6
100011 T1	*010000 T2	*000100 T3	000100 T4	000100 T5	000100 T6	000100 T7,B7
011000	011000	*000111 B3	000111 B4	000111 B5	000111 B6	000111 T8,B8
000111	000111	*011000 T9	*010000 T10	010000 T11,B11		
010100	010100	010100	010100	010100 T12	010100 T13,B13	
010110	010110	010110	010110 B10	010110 B12	010110 T14,B14	
100010	*000100 B2	*010000 B9	*011000 T15,B15			
111100	111100 T16	*100011 T17	100011 T18	100011 T19	*100001 T20,B20	
000100	*100010	100010	100010	100010	100010 T21	100010 T22,B22
100001	100001	100001	100001 B18	100001 B19	*100011 B21	100011 T23,B23
101100	101100	101100 B17	101100 T24,B24			
110111	110111	110111 T25	110111 T26	*110000 T27	110000 T28	110000 T29,B29
110011	110011	110011	110011	110011 B27	110011 B28	110011 T30,B30
110000	110000	110000	110000 B26	*110111 T31,B31		
010000	*100011	*111100	111100 T32	111001 T32,B32		
111001 B1	111001 B16	111001 B25	111001 B32	111100 T33,B33		

16.

属性＼分類法	Shell法	基数法	互換法	基数交換法
各判定毎に見出し語のすべてのビット(または文字)を検査する	✓		✓	
余分に大きな記憶域を必要とする		✓		
すでに表が分類されているときただ1回のパスで分類できる			✓	
分配分類法の1つである		✓		✓
分類のためには項目が2進数で表現されていなければならない				✓

17. これはコアと記号表の容量などの機械の制限に依存する．問題点は：

a. 表に登録する回数が多い場合，表を並べ換えるオーバヘッドが大きい．

b. 表の参照回数が多い場合，並べてない表を探索するのには時間がかかる．

コア容量が大きければ表の並べ換えが不要であるのでハッシュ法を使用するのが良い．記号の個数が少なければ，逐次探索法が最も簡単で最も能率が良い．

18. a. 記号表の大きさ ＜100：比較分類法を用いる．この程度の表の大きさのときはどの方法でも速度に変わりがない．比較分類法は余分の記憶容量が不要である．

記号表の大きさ ＞100：分配分類法を用いる．いずれにせよページのスワッピングが必要であるから，余分の記憶容量の問題はさほど重要でない．分配分類法の処理の方が速い．

b.

		場合 1	場合 2	場合 3	場合 4
		表の大きさ <100 表はそんなに頻繁に参照されない	表の大きさ <100 表は頻繁に参照される	表の大きさ >100 表はそんなに頻繁に参照されない	表の大きさ >100 表は頻繁に参照される
表の参照法	逐次探索法	X			
	2分探索法		X		
	ハッシュ探索法			X	
	2分法付きハッシュ探索法				X
表の管理法	順次記入	X			
	順次記入後互換分類を行なう				
	順次記入後基数分類を行なう				
	順次記入後 Shell 分類を行なう		X		
	ハッシュ法で記入			X	
	ハッシュ法で記入、欄がぶつかれば分類を行なう				X

19. a.

22000 MULHERN*b*

22008 HARRIS*bb*

22016

22024 MARY*bbbb*

22032	
22040	
22048	
22056	NANGLE*bb*
22064	MITCH*bbb*
22072	MADNICK*b*
22080	
22088	
22096	
22104	DONOVAN*b*
22112	FREYBERG
22120	
22128	
22136	
22144	
22142	

DONOVAN—	(4+6+5+6+4+1+5+0)	MOD	19	=	13
MULHERN—	(4+4+3+8+5+9+5+0)	MOD	19	=	0
FREYBERG—	(6+9+5+8+2+5+9+7)	MOD	19	=	13
MITCH——	(4+9+3+3+8+9+0+0)	MOD	19	=	8
HARRIS——	(8+1+9+9+9+2+0+0)	MOD	19	=	0
MARY——	(4+1+9+8+0+0+0+0)	MOD	19	=	3
MADNICK—	(4+1+4+5+9+3+2+0)	MOD	19	=	9
NANGLE——	(5+1+5+7+3+5+0+0)	MOD	19	=	7

b.　この方法は順に空いた場所を探すので，密度の高い表では効率が悪い．しかし，なお悪いことには1つの欄が削除されると処理全体が破壊される．たとえば，見出し語A，CおよびEに対して番地130が対応したならば各項目は次のように格納される：

129
130　A
131　C
132　E
133

もしCが削除されるとEの処理に問題が生じる.

表の主領域とは別に設けたあふれ領域を用いることもできる. この場合も詰め合せに問題がある.

見出し語から別の番地を計算する方法もある. しかし, 計算に時間がかかる.

同一の‘計算された番地’を持つ変数同志をポインタを用いて連鎖することが比較的容易である. 検出はより容易である.

20. a. Sell 分類法の泡出しは：

d=11, 6, 3, 2, 1
ASHTON, BEN
BERMAN, HARRIS
BRUNK, GLEN
DODSON, ORVILLE
DONOVAN, CAROLYN
DONOVAN, MAUREEN
DONOVAN, JAMES
DONOVAN, REBECCA
DONOVAN, MARILYN
DONOVAN, JOHN
FREYBERG, DUTCH
GOODMAN, LEONARD

HAMMER, MICHAEL
JACK, MARTIN
JOHNSON, JERRY
KOHN, NORMAN
MADNICK, STUART
NANGLE, ELLEN
RAMCHANDANI, CHANDER
SINNOT, MARY
ZILLES, STEVEN

b. I=6

c. 名前（last name）に対する2分探索法.
余分の記憶容量が不要. 名前が見付かると次に逐次探索法で姓が探される. この探索法は比較的遅いが余分の記憶容量が不要.

d. この問題に対しては組込むのが簡単であり, 表の大きさと処理速度の調和が良くとれている.

e. 大きな名前表のとき, 2分探索法は能率が良い. しかし名前に対してたくさんの姓が対応しているとき, 姓を探す逐次探索法は非常に遅い.

第4章　マクロ言語とマクロ・プロセッサ

1. 入力と出力はテキストである. MACRO, AIF および MEND のマクロ擬似命令以外の文字列は特に問題ではない. ほとんど任意の書式を処理することができる.

2. a. **マクロ定義表 MDT**

1.	XYZ	&A
2.	ST	1, &A

3.	MEND	
4.	MIT	&Z
5.	MACRO	
6.	&Z	&W
7.	AR	4, &W
8.	XYZ	ALL
9.	MEND	
10.	ST	&Z, ALL
11.	MEND	
12.	HELLO	&W
13.	AR	4, &W
14.	XYZ	ALL
15.	MEND	

マクロ名表 MNT

1.	XYZ	1
2.	MIT	4
3.	HELLO	12

展開されたアセンブラ語のテキスト

```
PROG     START
         USING    *, 15
         ST       2, 3
         AR       4, YALE
         ST       1, ALL
YALE     EQU      5
ALL      DC       F'3'
         END
```

b.　マクロ定義表 MDT

```
1.                 EXPO       &EXP
2.                 LCLA       &N
3.      &N         SETA       &EXP
4.                 AIF        (&N EQ 1).STOP
5.                 MR         0,2
6.      &N         SETA       &N-1
7.                 EXPO       &N
8.      .STOP      ANOP
9.                 MEND
```

マクロ名表 MNT

```
1.      EXPO       1
```

展開されたアセンブラ語のテキスト

```
EXPON   START      0
        USING      0,15
        L          2,BASE
        L          1,BASE
        SR         0,0
        MR         0,2
        MR         0,2
        ST         1,ANS
        BR         14
ANS     DS         F
BASE    DC         F'5'
        END
```

3.

	パス1で作成,変更しなければならない	パス1で参照されるが変更されない	パス1で作られパス2のために保存される	パス2で変更されない	パス2で作成，変更される	パス2で参照される
記号表	✓		✓	✓		✓
リテラル表	✓		✓	✓		✓
ベース・レジスタ表					✓	✓
一致テープ	✓		✓	✓		✓
MNTを含む機械命令表（MDT）	✓			✓		✓
擬似命令表		✓		✓		✓
入力デック		✓		✓		✓
マクロ定義表（MDT）	✓					
ロケーション・カウンタ	✓				✓	✓
マクロ・スタック	✓					

4. 本問の内容に関しては第1章の開いたサブルーチンと閉じたサブルーチン（単に**サブルーチン**という）の項で論じた．サブルーチンが呼出されると制御は呼出し者から別の手続きに移される．マクロ展開においてはコードがプログラムに挿入される．サブルーチン呼出しは実行時に行なわれ，マクロ展開はアセンブル時に行なわれる．何回も呼出されるルーチンをマクロにすると領域が多く必要であるが呼出し手順のオーバヘッドがない．一方，サブルーチンはこの手順に入るためのオーバヘッドがかかるが，ルーチンはただ1つの写しがあればよい．

スタック・フレームに相当するものはスタックのプッシュ・ダウンまたはLIFO (last-in／first-out) であり，回帰的なルーチンの中間結果を納めている．このことは第8.4節　回帰呼出しにて論じられている．

5. a. 作られない．入れ子になったマクロは，すべての外側のマクロ定義と同様に，MDT に登録される．

b. 変更されない．

c. 可能．外側のマクロが呼出された後で内側のマクロ定義が出てきたと

き，このマクロが定義される．この時点で MNT と MDT への登録が行なわれてマクロの呼出しが可能となる．

d. 不可能．内側のマクロ定義が行なわれるまでは MDT および MNT に登録されておらず，定義される前にはこれを呼出すことはできない．

e. できない．回帰的なマクロは内側にマクロ定義を含まない．しかし，このようなマクロの処理にはスタック・フレームが必要である．

f. 自分自身を呼出すマクロと，保存しておく必要のある中間的な変数の値．

6. 次のように引数として与えられる名札には何ら問題がない：

```
          MACRO
          ADD       &LAB, &LOC
&LAB      A         1, &LOC
          MEND
```

しかし次のようなときには名札がマクロ呼出しの度に挿入されるので記号の多重定義となる．マクロ・プロセッサはマクロが使用される度に新しい記号を生成しなければならない：

```
          MACRO
          ADD       &LOC
LABEL     A         1, &LOC
          MEND
```

7. a. すべてのマクロは呼出されるより前に定義されていなければならない．

b. 一般に，1パス・マクロ・プロセッサでは条件付きマクロ擬似命令の処理が可能である．きわめて一般的な条件処理が必要なとき（たとえば本問のような場合）には余分のパスが必要となる．

c. 余分のパスを設けるか，または，値の割付けられていない変数の判定を制限するような条件を課す．

8. a. （SR）

```
                    MACRO
          &LAB1     SR        &OP1, &OP2
          &LAB1     SET       LC
```

```
                BYTE      1B(16,8)
LC              SET       LC+1
                BYTE      &OP1(10,4),&OP2(10,4)
LC              SET       LC+1
                MEND
```

b. これは難しい問題である．しかしマクロの機能を示す良い例である．1つの解はL　1,900 のような文を受理し，次のような機械語を生成できるマクロを定義することである：

58	1	0	0	900
命令	R1	X2	B2	オフセット

ベース・レジスタを指定できるようにするともっと手間がかかる．

まず，変数 LC を初期化する．またベース・レジスタが必要であるから変数 BASE も初期化しておく．このために次のようなマクロ START を定義する：

```
        MACRO
&LBL    START     &LOC
LC      SET       &LOC
&LBL    SET       LC
ORIGIN  SET       LC
BASE    SET       0
        MEND
```

ベース・レジスタに値を設定するためには次のようなマクロ USING を定義する：

```
        MACRO
&LBL    USING     &LOC, &REG
BASE    SET       &REG
ORIGIN  SET       &LOC
&LBL    SET       LC
        MEND
```

上のマクロは条件付き擬似命令を用いずにベース・レジスタを割付け，それに値を設定する．ただしベース・レジスタは1個しか使用できない．

加算命令——RX形式

インデックス・レジスタは最後の引数で指定される．たとえば，インデックス・レジスタとしてレジスタ4を使用するとき命令はA　1,908,4のように書かれる．マクロ・プロセッサはこの命令を受理し，次の機械語を生成する：

				12ビット
5A	1	4	BASE	908
0	8	12	16	20　　31
命令	R1	X1	B1	オフセット

ここで 1, 4, 908は2進数で表現され，BASE は USING 命令で指定されているとする．

次のマクロ定義によりA命令が定義される．オフセットを計算して12ビットの値を生成する手順に注意してほしい．

```
        MACRO
&LBL    A       &REG, &DATA, &INDEX
&LBL    SET     LC
        BYTE    5A(16, 8)
LC      SET     LC+1
        BYTE    &REG(10, 4), &INDEX(10, 4)
LC      SET     LC+1
        BYTE    BASE(10, 4), &DATA／256-ORIGIN／
                  256(10, 4)
LC      SET     LC+1
        BYTE    &DATA-ORIGIN(10, 8)
LC      SET     LC+1
```

```
          MEND
```

インデックス・レジスタの指定がないときは値0を入れる．インデックス指定のためにかっこを使用するともっと手順が複雑になる．

他の命令コードを処理するときオフセットとベース・レジスタを計算し格納するためにマクロ LOC を定義しておく．

```
          MACRO
          LOC       &DATA
          BYTE      BASE(10,4), &DATA/256-ORIGIN/
                        256(10,4)
LC        SET       LC+1
          BYTE      &DATA-ORIGIN(10,8)
LC        SET       LC+1
          MEND
```

マクロLは次のように定義される：

```
          MACRO
&LBL      L         &REG, &DATA, &INDEX
&LBL      SET       LC
          BYTE      58(16,8)
LC        SET       LC+1
          BYTE      &REG(10,4), &INDEX(10,4)
LC        SET       LC+1
          LOC       &DATA
          MEND
```

9. a. 1) すべてのマクロ処理をアセンブラのパス1にて行なう．

2) LCLA および GBLA を導入すると定義段階を変更しなければならないが，展開段階には影響しない．

SETA は定義段階の変更なしに MDT に登録される．展開段階は変更が必要である．

b. GBLA 記号や LCLA 記号はただ1つの値を割当てられるので，このようにする必要はない．いったん値が更新されると古い値は不要になる．したがって，スタックに前の値を保存しておく必要はない．

これらの記号のためにはマクロ記号表 MST を作って記号の値を記入する方法が考えられる．この方法はスタックよりずっと直接的である．SETA 文は MST に登録された記号の値を更新する．

マクロ記号表　MST

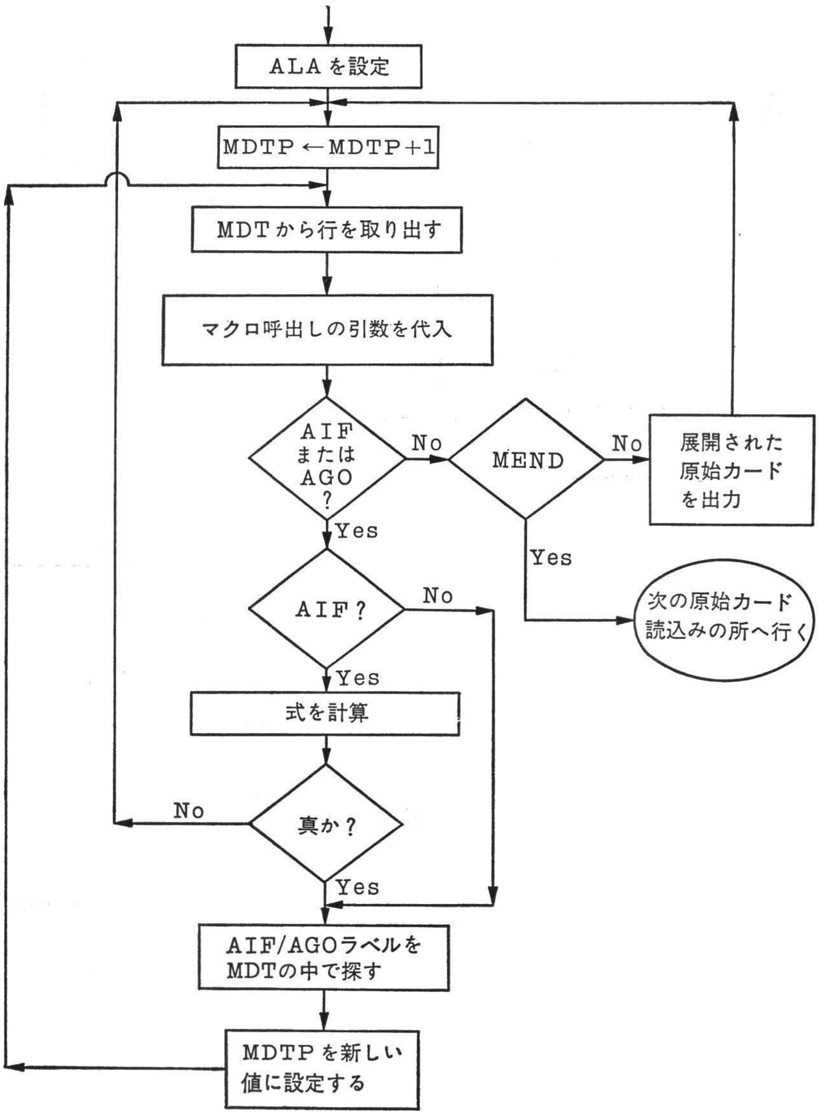


c. SETA 文の名札部の記号を MST の中で探す．見付かれば SETA 文の使われたマクロの名前がこの記号のマクロ欄に入っているか調べる．次にポインタが指している番地の内容を変更して SETA 文に指定された値とする．GBLA および LCLA 記号の処理には何ら差がない．

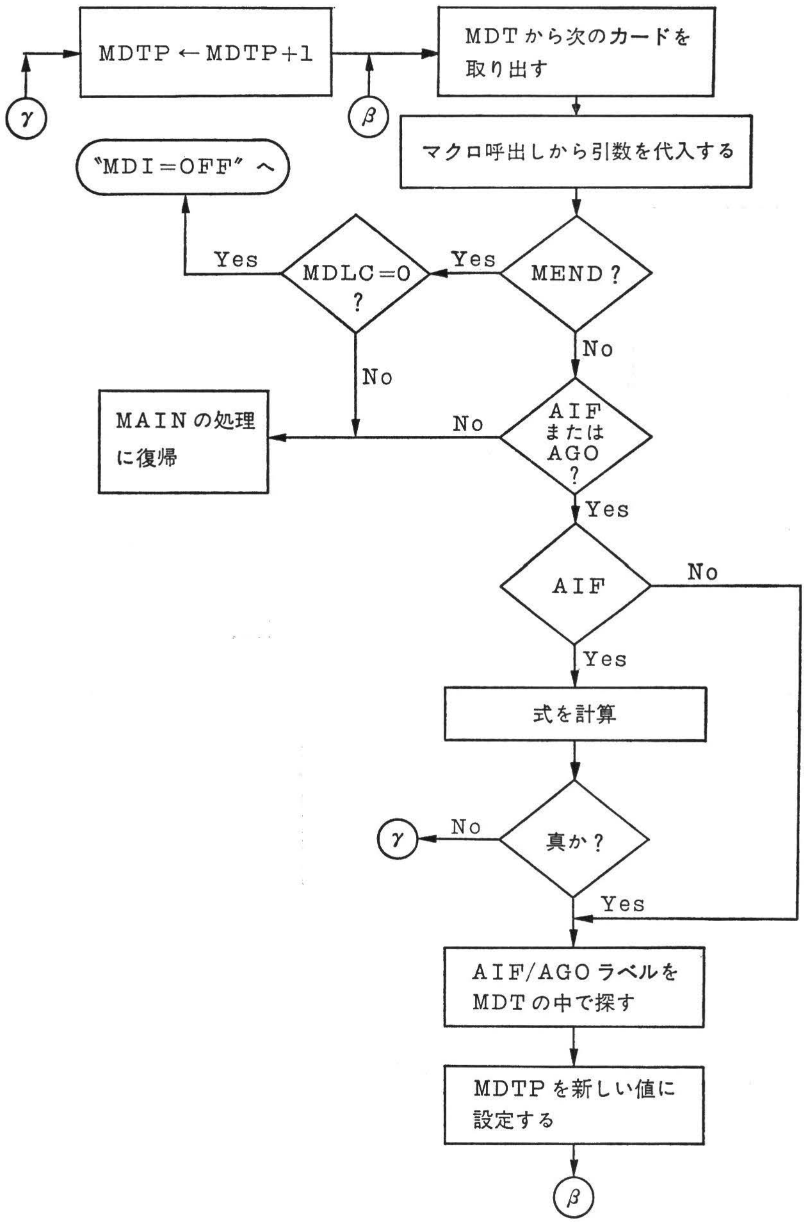
d. 1) LCLA 文は MST の1つの欄の LCL／GBL 表示を LCL とする．そしてこの記号に新しいポインタの値を割当てる．

2) GBLA 文は MST の1つの欄の LCL／GBL 表示を GBL とし，表を探して同じ記号で GBL 表示のあるものが存在すれば，ポインタの値をそれと同じにする．存在しなければ新しいポインタの値が割当てられる．

10. i) 2パス・プロセッサ（**図 4.2**）を変更する．



ii) 1パス・プロセッサ（**図 4.4**）を変更する.



iii) アセンブラの変更は，MDTP を S(SP+1) と変更する以外は1パス・プロセッサと同様である.

第5章 ローダ

1.

```
              EXTRN     SQRT
                .
                .
                .
    LABEL     AR        5,2
                .
                .
                .
    LOOP      B         LABEL
                .
                .
                .
              END
```

a. A (LABEL) または A (LOOP)

b. A (LOOP-LABEL)

c. A (SQRT+LOOP)

2. a. 単純再配置型：番地はローダ（リンケージ・エディタ）によって決定される.

b. 絶対値型　　：アセンブラは相対値 KOHN から相対値 LOOP を減じて絶対値を生成し，これに絶対値 ONE を加える.

c. 単純再配置型：PLACE の相対番地はアセンブラによって定められる. ローダはこの値にプログラムの開始番地を加える.

d. 複雑再配置型：アセンブラは番地定数の値として相対番地 LOOP の値の符号を負にして設定する. ローダはプログラムの開始番地を減じて—（LOOP の絶対番地）の値を計算してから SIN の値を加える.

3. a. “結合”はセグメント間の参照を解決する手順である. 例として問2

を見よ.

b. 1) ロード時に値がトランスファ・ベクトルに設定される.

2) ロード時に EXTRN の値が定められる.

3) アセンブル時にプログラマまたはアセンブラによって番地が定められる.

4&5) 実行時に，実際に参照が行なわれたときに，行なわれる.

6) オーバレイ構造がわかっているのでロード時に外部参照を解決することができる.

7) これは幾つかのセグメントを結合して1つの再配置可能なロード・モジュールを作成するバインダである. 結合はアセンブル（またはコンパイル）の後でローディングの前に行なわれる.

c. 1) ロード時に結合する方法ではプログラマは絶対番地を指定しなくてもよい. したがって, プログラマの負担が減ぜられ, プログラムのロード番地を変更するために再アセンブルをしなくてもよい.

2) 方式がより複雑で，より大きなローダが必要である. また処理時間も長い.

d. 1) プログラムは必要になるまで結合されない. 番地はプログラム時ではなく，実行時に計算して定められる.

2) この方式は特別のハードウェアを必要とし，処理時間のオーバヘッドが大きい.

4. RLD——どの番地の内容を再配置しなければならないか，またどのような処理が必要かの情報を含む.

ESD——結合のために必要なすべての情報を含む.

TXT——セグメントの目的コードを含む.

END——ロードすべきデックの終了を示す.

5. ESD

TXT

RLD

END

ESD は汎用外部記号表（GEST）を更新し，この表を用いて外部参照を結

合しながら TXT カードをロードして，RLD の情報で再配置を行ない，END がくると処理を終了する．

6. 可変長の文字列で名前を示すのに対して ID 番号は長さが一定である．局所記号 (LD) はローダで処理する必要がないので，これらのためには RLD カードが作成されない．したがって ID 番号を割当てる必要もない．

7. a. RLD カードが不要となる． 再配置の処理はベース・レジスタにより行なうことができる．

b. パス1の目的は外部記号表を作成することであるので，これ以上簡単にする所がない．しかし，パス2ではプログラムを再配置する必要がなくなる．ローダは単に目的コードが絶対型であるとみなしてロードするだけでよい．

8. a.

PGA	長さ	16
PGA1	オフセット	8
PGB	長さ	20
PGB1	オフセット	0
PGB2	オフセット	8
PGB3	オフセット	16
PGC	長さ	16
PGC1	オフセット	4
PGC2	オフセット	12

2語境界の条件により PGC は PGB の終りから4バイト空けて置かねばならない．

PGA	400
PGA 1	408
PGB	416
PGB 1	416
PGC	440
PGC 1	444
PGC 2	452

b. PGB 2 と PG 3 は局所記号である．

c.

番地	内容
400	400
404	420
408	8
412	12
416	440
420	416
424	420
428	0
432	450
436	XXXX
440	12
444	440
448	452
452	448
456	XXXX
460	XXXX

9. a.　パス1はすべての ESD カードを調べて汎用外部記号表 GEST を作成する．

b.　パス2はロード，再配置および外部参照の結合を行なう．

c.　本文で述べたような1パス・ローダを組込むためには，すべての外部記号は，先にロードされたプログラムで定義されていなければならない．このようなプログラムの順序を定められないことがある．別の方法として先にロードしたプログラムの中に値を設定する手続きを作ることもできる．しかしこの方法は先にロードしたセグメントを再度処理することがであるので真の1パス方式ではなく，$1\frac{1}{2}$パス方式とでもいうべきである．

d.　2パス DLL と全く同じ形式である．

e.　**図5.24**および**図5.25**のローダの流れ図に同じ．

（アセンブラ言語を制限して，たとえば先にロードされた手続きへの

外部参照のみ許すなどして，問c，d，eに答えてもよい.）

10. a.

```
      PGM     START    0
              EXTRN    SQRT
              EXTRN    TAN
              EXTRN    COS
              DS       5F
      ENT1    DS       5F
      ENT2    DS       10F
              DC       A(TAN+COS-ENT2)
              DC       A(COS)
              DC       A(ENT1-SQRT)
              END
```

b. 解は一意的に定まらない. 次のような小異が残る：

```
      ENT1    DS       5D
```

の所が

```
      ENT1    DS       5F
```

11. a. ABS カードを START カードの代わりにデックの先頭に置かねばならない. 絶対記号表を作成するためにロケーション・カウンタを適当な値に設定しなければならない.

b. ロケーション・カウンタを各パスの開始時に初期化する必要があるので，両方のパスで処理される. また再配置型のコード生成を抑制しておかねばならない.

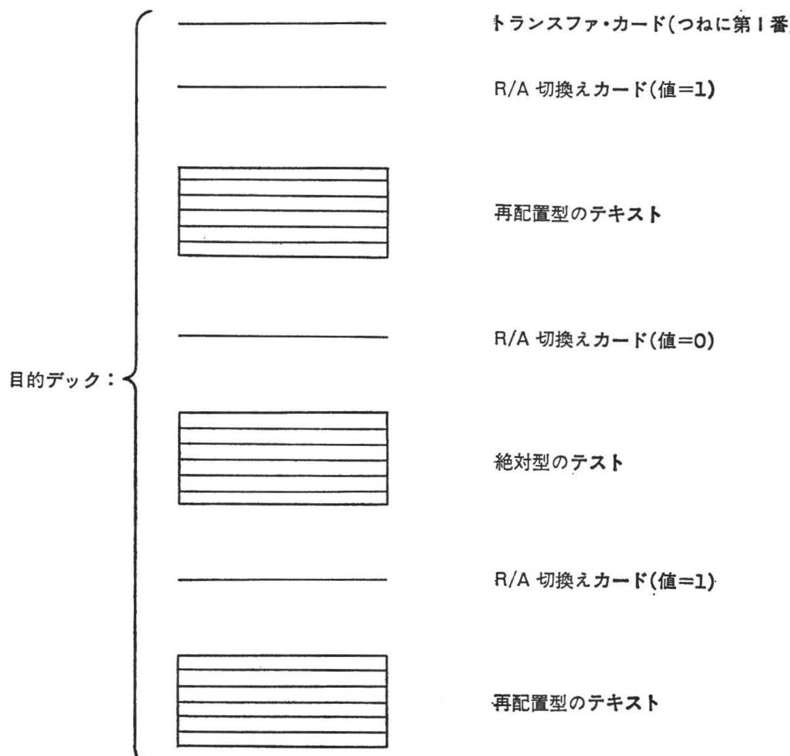
c. データ・ベースの書式は変更しなくてもよい.

b. 再配置／絶対値フラグは再配置型コードの出力を抑制する. パス2の開始時に， ABS 擬似命令によって記号表の中のすべての再配置／絶対値フラグを絶対値として設定するように組込めばよい.

12. a. 絶対型ローダに再配置／絶対値切換えカードを追加する.
このカードによりローダの内部の変数を設定して絶対型の処理か再配置型の処理かの区別を行なう.

カードの欄番号	内　容
1	カードの種類＝2（再配置／絶対値切換えカード）
2	情報のバイト数＝0
3	0：　絶対型の処理を行なう
	1：　再配置型の処理を行なう
4—72	空白
73—80	カード番号

b.　再配置／絶対値切換えカードの後にくるカードはその指定にしたがって処理される．この処理は次の切換えカードがくるまで続けられる．



13. a.　セグメント SOLN の目的デック

ESD

記　号	型	ID番号	相対番地	長　さ
SOLN	SD	01	0	50
A	ER	02	—	—
C2	ER	03	—	—

TXT

命令コード	相対番地	番地定数	番地定数の実際の値
BALR	0	—	—
SR	2	—	—
SLL	4	—	—
L	8	—	—
BR	12	—	—
DC	16	A (A+10)	10
DC	20	—	—
DC	24	A (C2)	0
DC	28	—	—
DC	40	—	—

RLD

ID番号	長　さ	フラグ(+/−)	相対番地
02	4	+	16
03	4	+	24

セグメント GRADER の目的デック

ESD

記　号	型	ID番号	相対番地	長　さ
GRADER	SD	01	0	60
C2	LD	—	24	—

TXT

命令コード	相対番地	番地定数	番地定数の実際の値
BALR	0	—	—
SR	2	—	—
L	4	—	—
ST	8	—	—
LA	12	—	—
L	16	—	—
BR	20	—	—

	DC	24	A (GRADER)	0
	DC	28	—	—

	ID番号	長さ	フラグ(+/−)	相対番地
RLD	01	4	+	24

b. 1) 可能な限り外部参照を結合する．各プログラムをつないで1つのTXT カードを作り，相対番地は複合セグメントの先頭からの番地として計算する．

2) SD は LD に格下げされる．

3) 新しい ESD に適合するように ID 番号を修正し，TXT の相対番地部を複合 TXT に適合するように変更しなければならない．

4) セグメント STUDENT の目的デックにおいて DELTA に対する ESD カードは依然として外部記号を参照している．

5&6)セグメント NEW の複合目的デック

	記　号	型	ID番号	相対番地	長　さ
ESD	NEW	SD	01	0	188
	STUDENT	LD	—	0	—
	A	LD	—	56	—
	SOLN	LD	—	72	—
	DELTA	ER	02	—	—
	C2	LD	—	152	—
	GRADER	LD	—	128	—

	命令コード	相対番地	番地定数	番地定数の実際の値
TXT	DC	40	A (A+10)	66
	DC	56	A (DLTA)	0
	DC	64	A (−SOLN)	−72
	DC	88	A (A+10)	66
	DC	96	A (C2)	152
	DC	152	A (GRADER)	128

	ID番号	長　さ	フラグ(+/−)	相対番地
	01	4	+	40

RLD	02	4	+	56
	01	4	−	64
	01	4	+	88
	01	4	+	96
	01	4	+	152

7) 2パス——パス1は ESD を調べる．パス2は TXT を変更し，新しい ESD と RLD を作成する．

8) 入力セグメント，プログラム・ライブラリ，局所記号表，複合 ESD および複合 RLD．

9) バインダの出力は再配置型の目的デックである． DLL の出力は実行形式のコア・イメージ・コードである．

10) 消去することはできる．しかしこれを行なう利益は小さい——このような場合は滅多にないし，処理のために計算時間がかかり，手続きが複雑となる．

14. a. **結合解除手続き** (de-linker) の入力は複合セグメントおよびこのセグメントの記号であるとする．

1) a) 長さの情報が残っていると明らかに LD 記号に格下げされた SD 記号の長さの部分を回復するのが容易である．のみならず，この方法は各セグメントが2語境界に置かれているので，元の長さを得るただ1つの方法でもある．

b) セグメントは2語境界に置かれているので，元のセグメント長ではなく， LD の相対番地を使用する必要がある．

c) 再配置はセグメント長でなく，複合 RLD を作成するときに記入したように，相対番地を使用する．

2) 元の長さは正確には決定できない．相対番地の引算で得た値は2語境界までの長さを与える．

3) 長さは不要である．その理由は，すべてのセグメントは2語境界に置かれるが，セグメント長が奇数語のときに必要となる詰め合せ用の1語以外は結合解除されたセグメントは元のセグメントと同一である．

b. 1) FROM 部は不要である．その理由は元の SD 記号と同じ記号を持っているからである．
IN 部は外部参照の場合必要である．
2) FROM 部は ER に使用できない．
IN 部は必要である．

c. 結合されたセグメント名の入力を省略することができる．
FROM 部を記憶するときには，LD 記号に格下げされた SD 記号についてはこの部分を空白とし，元の LD 記号についてはこの部分に記入して LD 記号であったことを表示する．

15. a. 明らかにすべての結合処理はバインダで行なうことができ，1つの再配置型のロード・モジュールが作成できる．また再配置処理もバインダで行なうことができる．この場合ローダは絶対型でよい．

b. (a)を参照．

c. 1パス．パス2で処理することは何も残っていない．

d. 再配置型，またはバインダが再配置を行なえば絶対型．
この方式は**リンケージ・エディタ**と呼ばれる（第5.1.7節参照）．

第6章　プログラム言語

1. a. コンパイル時に割付けられる．プログラムをロードしておく必要がなくなったときに解放される．

b. プログラムの制御にしたがって実行時に割付けられる．プログラムの制御または手続きの終了（END または RETURN）によって解放される．

c. 手続きに入ったとき割付けられる．手続きを終了したときに解放される．

d. (b)に同じ．

2. 割付けと解放は同じように行なわれる．しかし，被制御記憶域では最後に割付けたもののみが参照できる．一方，基底付き記憶域はポインタで指定して任意に参照することができる．

概念的には被制御記憶域は限定された基底付き記憶域の１つの形式であると考えられる．

3. DO ブロックはある意味で１つの文である．このブロックは THEN 節または ELSE 節の本体として使用できる：

```
IF A = B THEN DO;
          —
          —
          —
          —
          —
          END;
       ELSE   GO   TO   LABEL;
```

また DO ブロックは繰り返しの本体としても使用できる：

```
DO  I=1  TO  20  BY  2;
```

開始ブロックは，手続きブロックと同様にコードのブロックに含まれる変数の有効範囲を限定する．プログラムの実行の流れにしたがって開始ブロックが実行される．

手続きブロックは，外部で宣言された変数と無関係に，手続きの中で変数を定義する．また外部変数も参照可能である．通常のプログラム実行は手続きブロックを飛ばして進行する．手続きは呼出されたときのみ実行される．

これらは用途が異なっている．

4. DO グループは変数の有効範囲に何ら関係しない．DO は名札の有効範囲に関係する．すなわち，DO グループの内側の名札に制御を移すことはできない．このグループは DO 文によって始まり，END 文で終了する．（１回または繰り返し）．
DO グループが IF 文の THEN 節として使用されていれば ELSE 節は飛ばされる．それ以外のときは制御は次の文に移る．

手続きおよび開始ブロックが１つのブロックの内側にあるときは，内側の

手続きの中で外側の手続きの変数と同じ名前の変数を陽に宣言するとき以外，変数の有効範囲に影響しない．同じ名前を内側のブロックで宣言したときは外側で宣言された変数を内側のブロックで参照できない．

開始ブロックは BEGIN 文で始まり，それに対応する END 文で終了する．

CALL 文によって呼出された手続きに制御が渡される．関数手続きは代入文の中で暗に手続きを参照することによって制御が渡される．これらの実行は END 文または RETURN 文によって終了する．

5. 問題 6.4 で述べた制御のほかに， EXTERNAL と宣言するとこの記号は外部手続きから参照できる．INTERNAL と宣言するとこの記号は外部手続きからは参照できない．

プログラムは，異なったプログラマによって，名前の混同の心配なしに，幾つかに分割して作成することができる．

6. 1) 単に宣言の範囲にしたがって内部の手続きに引渡す．

2) 手続きまたは呼出し（または関数呼出し）のパラメータ表によって引渡す．

3) 両方の手続きで EXTERNAL 宣言を行なうことによって引渡す．

7.

実行された文	変更された値
(1)	X = −2
(2)	Y = −2
	W = 0
	Z = 0
(8)	M が割付けられる．
(9)	M = 4
(10)	Z = −4
(11)	
(12)	
(13)	W^1 = 2
(14)　(3)	
(4)	

	(5)	$X^1 = -4$
	(6)	Z = 14
(15)		
(16)		
(19)		
(20)		X = 0
(21)		
(22)		Z = 0
		Y = 0
		W = 0
(23)		

8.

実行された文		変更された値
(1)		
(2)		
(3)		Z = 12
(4)		
(5)		
(6)		X = 5
(7)		ZED = 17
(8)		
(9)		
(10)		X = 1
(11)		
(12)		
(25)	(37)	
	(38)	Z = 1
	(39)	Z = −39
	(40)	
(41)	(35)	
	(36)	X = 5

(37)

(38)　Z = 25

(39)　Z = −15

(40)

(42)　(26)

(27)

(28)

(29)　Z^1 が割付けられる．

(30)　X = 4

(31)　Z^1 = 16

(32)　X = 20

(33)　Z^1 が解放される．

(43)　(44)

(45)

(46)　STR = 60

(47)　Z = 80

(49)

第7章　形式的体系とプログラム言語

1. **a.** 1) 〈文〉::=〈節〉△〈文〉　右

2) 〈節〉::=〈節〉□〈人〉　左

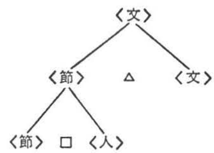
4) 〈語〉::=〈語〉$〈文字〉　左

6) 〈咳払い〉::=〈咳払い〉△〈文字〉　左

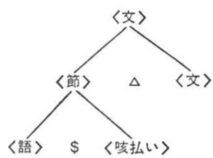
b. i) 〈文〉::=〈節〉△〈文〉

〈節〉::=〈節〉□〈人〉

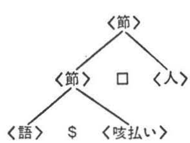
□は△より順位が高い．



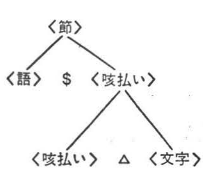
ii)　〈文〉::=〈節〉△〈文〉
〈節〉::=〈語〉$〈咳払い〉
$は△より順位が高い.



iii)　〈節〉::=〈節〉□〈人〉
〈節〉::=〈語〉$〈咳払い〉
$は□より順位が高い.



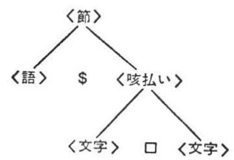
iv)　〈節〉::=〈語〉$〈咳払い〉
〈咳払い〉::=〈咳払い〉△〈文字〉
△は$より順位が高い.



v)　〈節〉::=〈語〉$〈咳払い〉

〈咳払い〉∷＝〈文字〉□〈文字〉

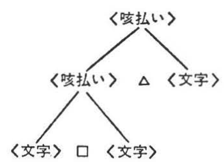
□は＄より順位が高い．



vi)　〈咳払い〉∷＝〈咳払い〉△〈文字〉

〈咳払い〉∷＝〈文字〉□〈文字〉

□は△より順位が高い．



c.　(b)をまとめると：

i)　□ ＞ △

ii)　$ ＞ △

iii)　$ ＞ □

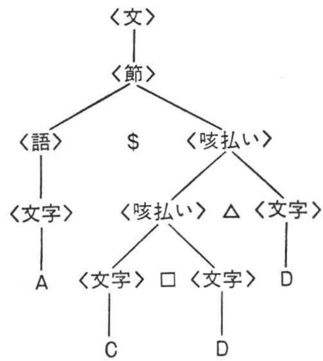
iv)　△ ＞ $

v)　□ ＞ $

vi)　□ ＞ △

（ii）と（iv）および（iii）と（v）は各々矛盾するように見えるであろう．しかし，（ii）の関係はこの書換え則によって生成されたストリングだけに適用される．また（iv）に示した逆の関係は別の書換え則を適用することによって生じる．これら2つから生成されるストリングは各々異っているのであいまいさや矛盾は起こらない．

d.



A $ C □ D △ D

註：□と△および△と＄の間の順位を保存するような樹を生成する方法は存在しない．

2. 1)　言語を記述する．

2)　言語に含まれるストリングを認識する．

3)　言語の複雑さを研究する．

4)　プログラムの等価性を示す．

5)　言語を比較する．

詳細は第7．1節を参照せよ．

3. **a.** 1)　**文法**とは言語を生成するための規則の集合である．

2)　**言語**とは構造と意味を有する記号のストリングである．

3)　**機械**または**オートマトン**とは言語に属するストリングを受理する装置である．

b. BNF 定義は文法である．正準体系も文法であるが BNF よりも記述能力が大きい．

c. BNF によって記述できる言語はつねに正準体系で記述できる．逆は必ずしも真ではない．正準体系は BNF が記述できない（文脈規定）言語の特徴をも記述できる．BNF は制限された正準体系の能力を持つ．

4. 生成文法では，与えられたストリングと一致するまで，または，このストリングが正しくないことを示すために同じ長さのストリングがつきてしま

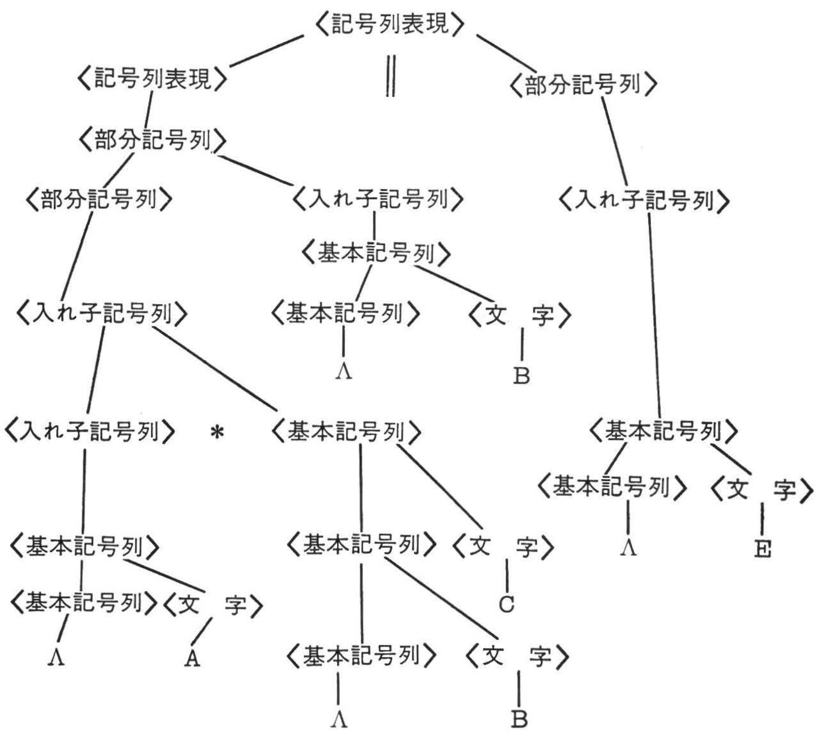
うまで，多数のストリングを生成しなければならない．この方法は時間がかかる．

5. **a.** BNF は言語の文脈規定の特徴を記述することができない．

b. 正準体系は文脈規定の特徴を記述するのに十分な'集合と関係'（たとえば，名札に対する**包含**の関係．第 7.6.1 節参照）を規定することができる．

6. **a.** たくさんの例があるがこれらの内の２つを示しておく．両方共文字の規則以外の各規則を少なくとも１回使用している．

i)

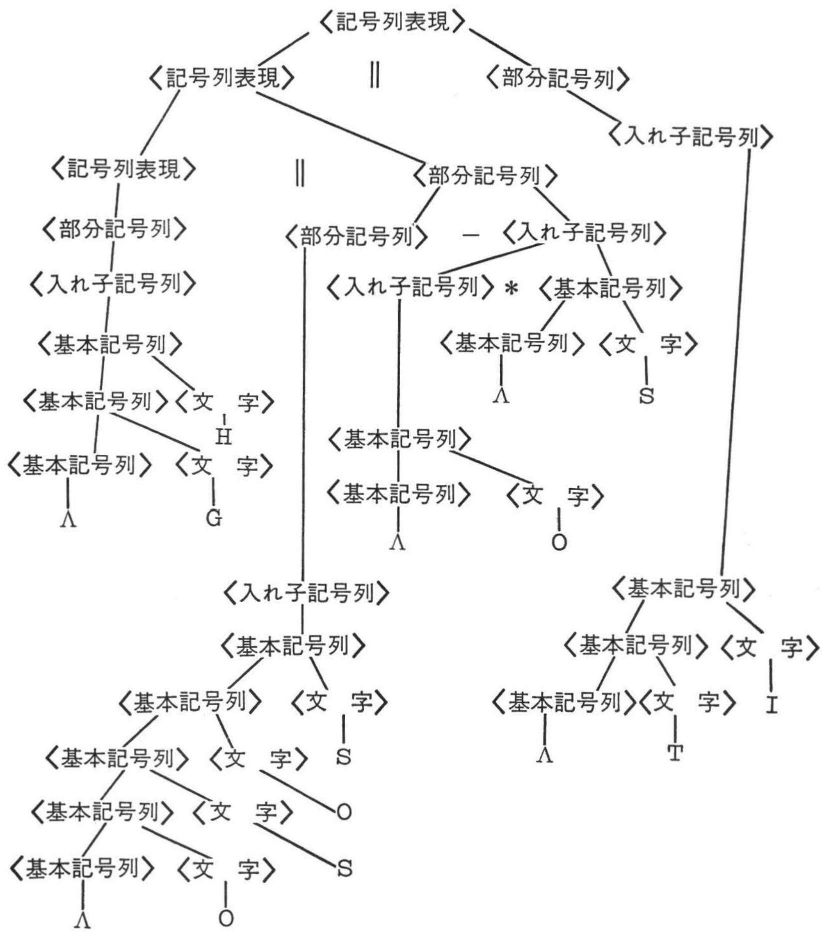


A∗BC－B‖E（＝BCABC－B‖E

＝CAC‖E

＝CACE）

ii）



GH‖OSOS−O*S‖TI

GH ‖ OSOS - O ∗ S ‖ TI

(＝GH ‖ OSOS - SOS ‖ TI

＝GH ‖ O ‖ TI)

＝GHOTI

これは "fish" と読む.

(GH ： LAUGH

O ： WOMEN

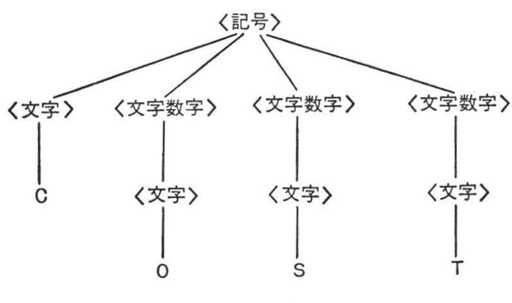
TI ： COMMOTION)

b.　b) ∗ > - > ‖

c.

左演算子 ＼ 右演算子	‖	∗	−
‖	<	>	>
∗	<	<	<
−	<	>	<

7.　**a.**　〈記号〉から1つの記号への導出はすべて次のように表わされる：

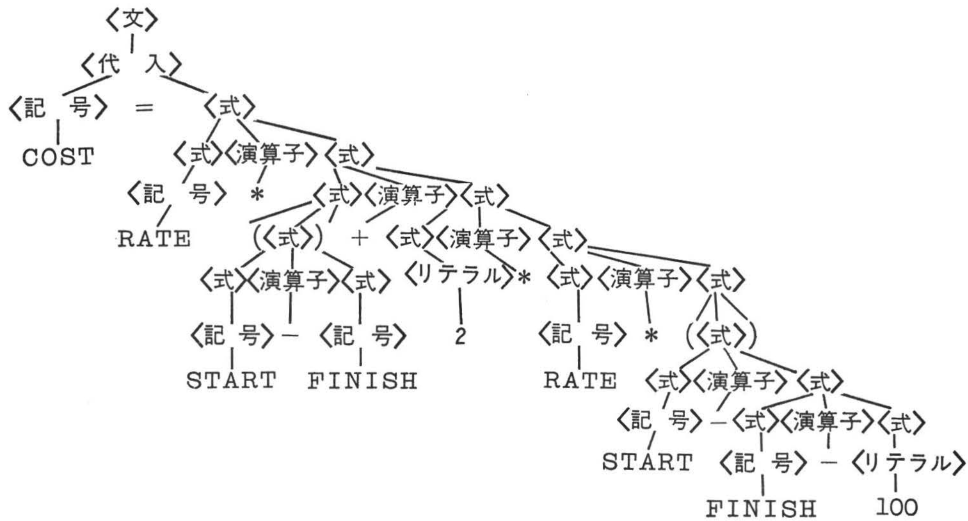


これを次のように省略して書くことにする：

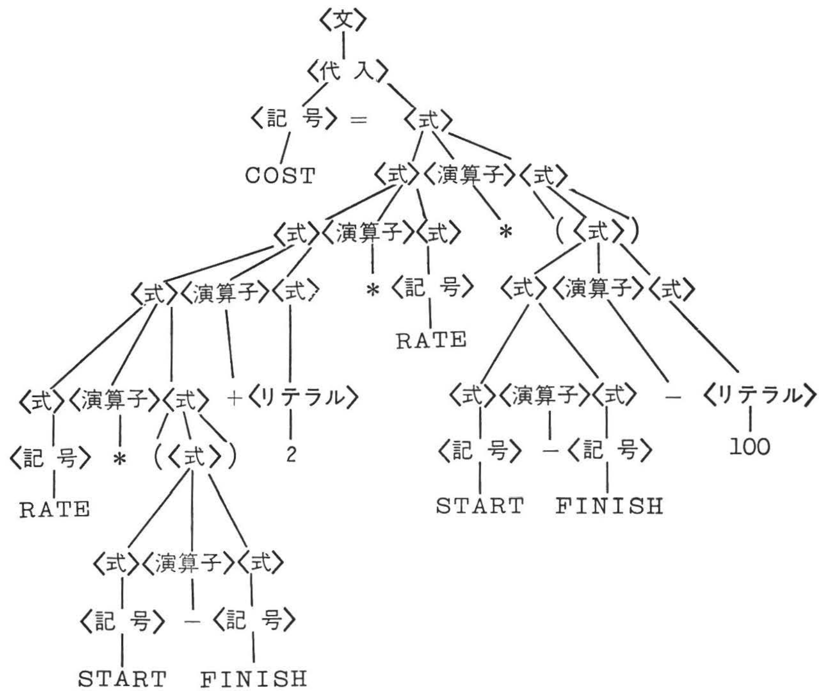


問の文に対しては多数の構文解析の樹が存在する．これらの内の2つを次に示す：

i）



ii）



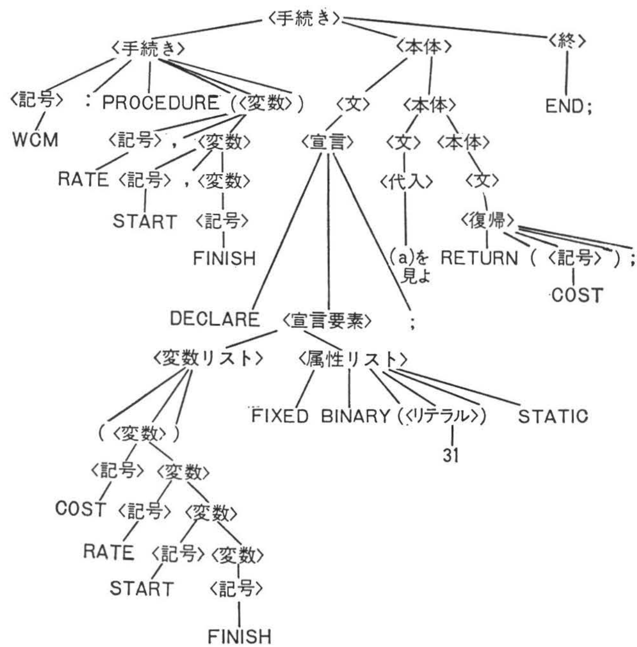
註：(ｉ) は次に等価である：

COST＝RATE＊((START－FINISH)＋(2＊(RATE＊(START－(FINISH－100)))))

(ii)は次に等価である：

COST＝((((RATE＊(START－FINISH))＋2)＊RATE)＊((START－FINISH)－100)

b.



c.　(a)で2とおりの導出を示したように，優先順位は定義されていない．かっこをつけた式は正しい．かっこをつけることによって始めて計算の順序をあいまいさなしに規定できる．

d.　次を削除する：

〈式〉::=〈リテラル〉|〈記号〉|〈式〉〈演算子〉〈式〉|(〈式〉)

〈演算子〉::=＋|－|＊|／|＊＊|＝

次を追加する：

〈きごう〉::=〈リテラル〉|〈記号〉|(〈式〉)

〈迷路〉::=〈きごう〉＊＊〈迷路〉|〈きごう〉

〈項〉::=〈項〉＊〈迷路〉|〈項〉／〈迷路〉|〈迷路〉

〈式〉::=〈式〉＋〈項〉|〈式〉－〈項〉|〈項〉

8. a. カノン6と8により，＊が＋より順位が高いと定義されている．

b. 次の変更を行なう：

5) **q因子⊢q項**

6) **t項；q因子⊢t＊q項**

次を追加する：

10) **P一次子⊢P因子**

11) **P一次子；S因子⊢P$S 因子**

9. a. 〈文字〉::＝P|Q|R|S

〈記号〉::＝〈文字〉|〈記号〉〈文字〉

〈一次子〉::＝〈記号〉|(〈式〉)

〈項〉::＝〈一次子〉|〈項〉＊〈一次子〉

〈式〉::＝〈項〉|〈式〉＋〈項〉

b. 等価な BNF は存在しない．

カノン 5, 6, 7, 8, 9 は BNF で定義できない．

各自，問10〜問12を解いてみよ．多数の解答が存在する．

第8章 コンパイラ

1. **図 8.13** を参照せよ．

2. （トランスレータとインタプリタ）

トランスレータは1つの言語のプログラムを入力し，他の言語のプログラムを出力する．インタプリタは1つの言語のプログラムを入力し，目的コードを生成せずにプログラムの意図したことを実行する．（たとえば IBM 370 は機械コードのインタプリタである．また BASIC や APL などの言語は通常インタプリタとして組込まれる．）

コンパイラの構文解析段階はインタプリタであると考えられる．すなわち，構文解析段階は変形則を解釈する．

註：変形則はトランスレータによってコンパイルすることができる．

3. **パスと段階**

パスとは入力デックやデータ・ベースを物理的に走査することである．段階とはコンパイラなどの論理的なセグメントであり，1つの機能を果たす．たとえば，最適化段階は何回もデータを走査（パス）するが，構文解析段階および解釈段階は1つのパスで組込むことができる．

構文解析と意味の解釈

構文解析は構文のみに関係する．意味解析は意味に基づく解析である．

トークンと不偏記号

トークンとは単語の単位であり，入力の一部である．不偏記号とは可変書式のトークンを固定書式で表現するためにコンパイラが生成した単位である．

4. **利点**：コンパイラはアセンブラの仕事（たとえば，記号表やリテラル表の作成）の多くをすませており，これらを2回繰り返す必要がないので高速である．

欠点：より大きな，複雑なコンパイラが必要となり，一時に必要なコア容量が増す．

5. 可変書式のデータよりも不偏記号の方が処理しやすい．必ずしも不偏記号を用いる必要はないが，処理段階の間で長さの異なる記号やデータを受け渡すようにすると手続きが複雑となる．

6. 各段階は論理的に独立な単位として考えることのできる独自の機能を果たす．コンパイラは1段階のプログラムであると考えることもできるが，このような表現法はコンパイラの構成法について何も示していない．同様に“行列生成”段階といっただけではコンパイラやその設計を他の人に説明するのには十分な内容を含まない．

7. 単語，構文，解釈および最適化の段階は全般的に見て機械に独立である．すなわち，IBM 360 で実行されるコンパイラは，これらの段階を変更し

なくても，他の計算機のためのコードを生成することができるであろう．これらの段階のデータ・ベースは機械に独立である．コンパイラの最後の3段階も機械に独立の記号表，リテラル表を使用しているという人もあるだろうが，これは正しくない．コード生成およびアセンブル段階は機械に関係するものである．

8. 否．これはコンパイル時間と実行時間の関係で定まる．たとえば，デバッグのためにただ1度だけ実行されるプログラムを長い時間かけてコンパイルするにはおよばない．希望する最適化の程度をパラメータで指定するコンパイラもある．

9. 生成則はストリングを生成する．変形則はストリングを構文解析して分解する．たとえば，BNFの生成則を用いてPL/Iプログラムを生成することができ，コンパイラの変形則を用いてPL/Iプログラムの構文解析を行なうことができる．

10.

	生成される表	固定の表	言語に関係する	機械に関係する	言語に独立	機械に独立[註]
終端記号表		X	X			X
記号表	X		X			X
リテラル表	X		X			X
変形則		X	X			X
不偏記号	X		X			X
行列	X		X			X
行列の演算子		X	X			X
コードの生成則		X	X	X		
活動ルーチン		X	X			X

註：実際には，これらのデータ・ベースは幾分かは機械に関係するものもあるし，書式やデータの内容は機械の影響を受ける．

11. コード生成段階とアセンブル段階を変更してH645のコードを生成する．

12. 解釈段階は行列の欄を作成する．

機械に独立な最適化段階は行列を変更（たとえば，欄を削除する）ことがある．

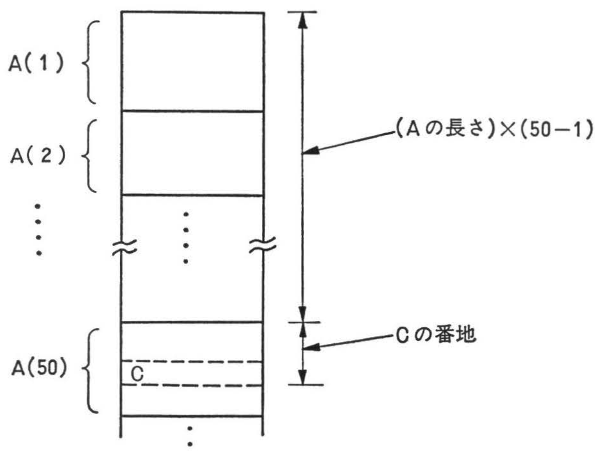
コード生成段階は修飾せずに行列を使用する．

記憶域割当て段階は記憶域に関係する欄（たとえば | STATIC | 10 | | ）を追加する．

13. Xの番地，精度およびデータの型

A.B.C(50) を参照するためにはCのデータの型，精度および番地が必要である．番地を求めるにはAの中でのCの番地とAの長さが必要である．所要の番地は次のように計算される：

Cの番地＋(Aの長さ)×(50－1)



14. ///*/ は //*/ とするのが正しい．
〈記号〉/″/*/2 は〈記号〉//″*/2
とするのが正しい．

15. 固定の表：　終端記号表
変形則
コード生成則
作成される表：不偏記号表
記号表
リテラル表
行列

16. 正準体系と BNF は言語の生成的定義である．
BNF は文脈規定言語の特徴を記述できない．

変形則はストリングの構文解析を行なう.

17. PROCEDURE 文は行列の中に次のような簡単な形の欄を生成する：

. procbgn 〈記号〉

RECURSIVE のような属性が指定されると，手続きの導入部で回帰呼出しの条件を設定するために，もっと複雑な行列の欄を作らねばならない.

DCL 文は直接には行列の欄を生成しない.
解釈段階は DCL 文を処理して記号表に適当な情報を記入する.
記憶域割当て段階では記憶域が割付けられるようにし，初期値の属性が指定されていれば初期化が正しく行なえるように用意する．このために記憶域割当て段階では，コード生成段階が記憶域を割付け，初期化を行なうようなコードを生成するように，適当な行列の欄を生成する.

18. ＋に対するコード生成を参照して解釈せよ.

19. レジスタ割当ての最適化.
無用なロード，格納命令の削除.
短く（RX 形式よりも RR 形式）かつ，速い（2を掛けるより1ビット シフトする）効率のよい命令の使用.

これらの型のコード最適化は機械に関係するものであり，コード生成段階で行なわれる．この最適化をこれ以前の段階で行なうことにするとその段階が機械に依存するものとなってしまう.

20. 最適化段階で行なわれる．その理由はこれが完全に機械に独立な最適化であり，行列を用いて行なうのが最も能率がよいからである.

21. **a.**

X			X
		X	
	X		X
X			X

b. 共通項の削除： 機械に独立な最適化段階で行列の欄を削除する.

コンパイル時の計算：機械に独立な最適化段階で行列の欄にリテラルを代入する.

ブール表現の計算： 機械に独立な最適化段階で不必要な演算をさけるような条件を行列の欄に代入する.

コードの展開： コード生成段階で呼出しに対してマクロ本体を代入する.

22. 2つの場合が存在する．1つは単項演算子（−）を持つ場合で，他は2項演算子（−）を持つ場合である.

場合 1：[注]

−(A+B)+X

B−A+X

−	B	A
+	M1	X

−	A	B
−	0	M1
+	M2	X

場合 2：

X−(A−B)

X+B−A

+	X	B
−	M1	A

−	A	B
−	X	M1

注：本文には単項演算子について何も述べていない．単項演算子の場合には，たとえば，補数ロードのような特別の命令を使用することを指定する符号を用いてもよい.

場合1：識別が問題である．行列の中で単項演算子（−）を持つ一対の欄を判定しなければならない．その後は本文で述べた手順にしたがえばよい.

場合2：計算の順序が関係するので場合1よりも難しい．この最適化は判定するのが難しい.

23. a. b. c. 種々のデータの型に対する算術演算を処理する機能を組込むのに幾とおりかの方法がある．１つは単語，構文および解釈段階を本文のものと同じにしたままで行なう方法である．ただし，活動ルーチンは記号表に FIX BIN 以外の変数のデータの型を登録する．コード生成段階は記号表の欄を用いて適当なコードを生成する．コード生成段階は演算数の不偏記号を記号表の指標として用い適当な欄を求める．

第２の方法は解釈段階を変更して，算術式の中の変数のデータの型を判定し，行列の中にデータの型に応じた演算数を入れるものである．たとえば，A+B においてAとBが浮動小数点数のとき次の欄を作る：

+FLOAT	A	B

コード生成段階では適当なマクロ定義を行なっておきコードの生成を行なう．

第３の方法では，行列に欄を追加して各演算のデータの型を記述することである．ただし，型の混合した式を認めない．これは活動ルーチンが行列の欄を作成するとき記号表／リテラル表からデータの型を判定することによって行なうことができる．この方法は変形則を使用しない．変形則の操作はストリングに対してのみ行なわれ，ストリングから型を決定することはできない．

d. 上述の方法によるとコード生成段階は例に示したように変更しなければならない．しかし，FIXED BINARY の展開と FLOAT BINARY の展開は類似しており，１つのマクロを用いて命令コードだけを変更するだけでよい．FIXED DECIMAL は全く異なった展開が必要である．

e. 混合モードの式のために行列には各演算数に対して“データの型”の

欄を用意することができる．コード生成段階において型の変換が必要か否かを決定する．もっと複雑なマクロ定義を行なう方法も考えられる．

24. 解 1) **a.** リテラル表

リテラル	**基数**	**尺度**	**精度**	**番地**
10	BINARY	FIXED	15	0

|←――単語段階の後――→|（リテラル～精度）

|←――記憶域割付け段階の後――→|（リテラル～番地）

記号表

名前	**基数**	**尺度**	**精度**	**記憶域のクラス**	**番地**
BEGIN	名札	手続き名			
A	BIN	FIXED	15	STATIC	0
B	BIN	FIXED	15	STATIC	2

|単語段階の後|（名前～基数）

|←――解釈段階の後――→|（名前～記憶域のクラス）

|←――記憶域割付け段階の後――→|（名前～番地）

b. ;

END

PROCEDURE

c. “*” はスタックとの一致がとれたことを示す．

数は図 8．17 の変形則の番号を示す．

1*	18	8
2*	19*	9
4*	7	10*
4	8	
5*	9*	
7*	12*	
14	7	

15*	8
15	9*
16*	12*
14*	7

d. 行列

1)	. procbgn	BEGIN	
2)	. arg	A	
3)	. arg	B	
4)	=	A	10
5)	=	B	A
6)	. procend		
	記憶域割付けのための情報		

e. 次の結合規約を仮定する：

i) 呼出し者プログラムへの復帰番地はレジスタに納められている.

ii) 返却されるべき値の番地を示すポインタはレジスタ14に設定されている.

			行列の欄から
BEGIN	START	0	
	STM	15, 13, SAVE	1
	BALR	15, 0	
	USING	*, 15	
	ENTRY	A	2
	ENTRY	B	3
	LH	1, =H' 10'	4
	STH	1, A	

```
        LH      1, A             } 5
        STH     1, B             }
        LM      15, 13, SAVE     } 6
        BR      15               }
A       DS      H                }
B       DS      H                } 記憶域割付けにて
SAVE    DS      15F              }
        LTORG                    } 6
        END                      }
```

f. 次のいずれかの方法を用いる：

1) 誤りを検出したとき活動ルーチン op_prec に error を呼出させる.

2) op_prec を変更して，誤りを検出したとき
〈結果〉〈誤り表示〉をスタックの先頭に置くようにする.
次の変形則を追加する：

12.5：〈結果〉〈誤り表示〉/error//7

解 2) a. **リテラル表**

リテラル	基数	尺度	精度	番地
10	BINARY	FIXED	15	0

|←――単語段階の後――→| (リテラル〜精度)

|←――記憶域割付けの後――→| (リテラル〜番地)

記号表

名前	基数	尺度	精度	記憶域のクラス	番地
MIT	名札	手続き名			
JOHN	BINARY	FIXED	31	STATIC	0
STU	BINARY	FIXED	31	STATIC	4
BILL	BINARY	FIXED	15	STATIC	8

単語段階の後 |

|←――――解釈段階の後――――→|
|←――――記憶域割付けの後――――――→|

b. ;
END
PROCEDURE

c. “＊”はスタックの先頭と一致がとれたことを示す.

1*	18*	7
2*	14*	8*
4*	18	5*
4	19*	7
4	7	8
5*	8	9
7*	9*	10*
14	12*	
15*	7	
15	8	
16*	9*	
14*	12*	

d. 行列

1)	. procbgn	MIT	
2)	. arg	JOHN	
3)	. arg	STU	
4)	. arg	BILL	
5)	*	STU	BILL
6)	=	JOHN	M5
7)	*	BILL	10
8)	−	JOHN	M7
9)	=	STU	M8
10)	. rtn		
11)	. arg	STU	
12)	. procend		
	記憶域割付けのための情報		

e.

行列の欄から

```
MIT     START   0                 }
        STM     15, 13, SAVE      } 1
        BALR    15, 0             }
        USING   *,15              }
        ENTRY   JOHN              } 2
        ENTRY   STU               } 3
        ENTRY   BILL              } 4
        LH      1, BILL           }
        SR      0, 0              } 5
        M       0, STU            }
        ST      1, M5             }
        L       1, M5             } 6
        ST      1, JOHN           }
        LH      1, =H '10'        }
        SR      0, 0              } 7
        MH      0, BILL           }
        ST      1, M7             }
        L       1, JOHN           }
        S       1, M7             } 8
        ST      1, M8             }
        L       1, M8             } 9
        ST      1, STU            }
        LA      14, STU           } 10, 11
        LM      15, 13, SAVE      } 12
        BR      15                }
```

JOHN	DS	F	記憶域割付けにて
STU	DS	F	
BILL	DS	H	
M5	DS	F	
M7	DS	F	
M8	DS	F	
SAVE	DS	15F	
	LTORG		12
	END		

註：効率のよい記憶域割当て段階は M5 および M7 に同じ番地を割付ける．これらは同時には参照されない．

25. a. 単語および構文段階である．単語段階は単語解析を完了するのに終端記号表の中の分割文字を利用することができない．すなわち，単語段階では記号表の欄を指す不偏記号によって DO 17 I を置換えるのがよいか，または，終端記号表の欄を指す不偏記号によって DO を置換えるのがよいか判断できない．

b. 段階の間に相互の関連を持たせる必要がある．単語段階はストリングを幾つかの方法で解析するように作られていなければならない．問の例では第1回の解析結果は次のようになる：

DO17I/＝/1/, /10/, /1

構文段階は構造上の誤りを発見すると，直ちに誤り処理を行なわずに，単語段階に戻る．単語段階は別の解析規則で次のように解析する：

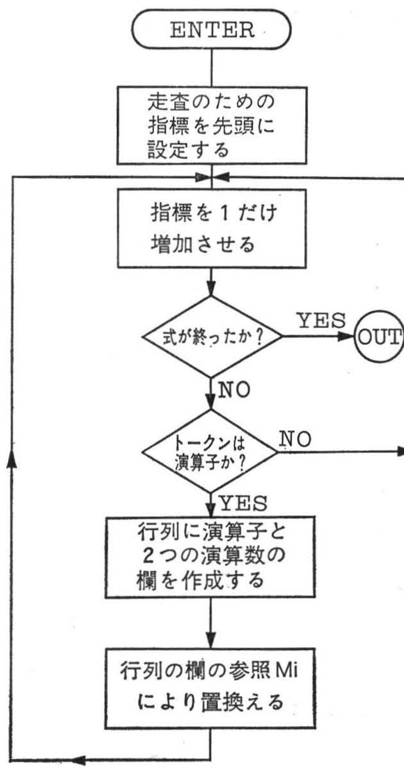
DO/17/I/＝/1/, /10/, /1

このような方法ではプログラマの意図に反して間違った解析結果が正しいとして受理されることが起こり得る．

26. (分岐や記憶領域に参照などで) 後で定義される記号を参照する問題 (for-

ward reference：前方参照）はコアの使用効率を犠牲にして解決しなければならない．この前方参照は，第5章の BSS ローダのように，トランスファ・ベクトルを用いて解決できる．最適化にはきわめて限定された技法のみが使用できる．また言語に対して，DCL 文は変数が使用されるよりも前に置かなければならないといった制限を課す必要が生じる．1パス方式では'論理的な機能単位の分割'ができない．手続きの中に機械に関係する部分と機械に独立な部分が混在するであろう．

27. a. かっこは使用しないものとする．



b. 〈記号〉＝//***/

1: 〈任意〉〈演算子〉〈記号〉/mat_ent/*/

〈任意〉; ///2

//*/1

2: 〈記号〉=〈任意〉;/assign/''/

活動ルーチン:

mat_ent——次の欄を行列の中に生成し，スタックの中の〈任意〉〈演算子〉〈記号〉を行列の欄への参照 **Mi** で置換える.

+〈任意〉〈記号〉または
-〈任意〉〈記号〉

assign——次の欄を行列の中に生成する.

=〈記号〉〈任意〉

c.
```
PARSE : PROCEDURE (EXPRESSION);
DECLARE 1 MATRIX (100) EXTERNAL,
            2 OPCODE CHAR(1),
            2 OPI CHAR(8),
            2 OP2 CHAR(8),
         MAT_INDX FIXED BIN EXTERNAL,
         SCAN EIXED BIN INITIAL (1),
         1 EXPRESSION BASED(P),
           2 SIZE FIXED BIN,
           2 TOKEN (J REFER (SIZE)) CHAR (8);
LOOP : SCAN=SCAN+1 ;
       IF SCAN>SIZE THEN RETURN ; ELSE ;
       IF TOKEN (SCAN)='+' |TOKEN (SCAN)='-'
         THEN DO :
               OPCODE (MAT_INDX)=TOKEN(SCAN) ;
               OP1 (MAT_INDX)=TOKEN (SCAN-1) ;
               OP2 (MAT_INDX)=TOKEN (SCAN+1) ;
               TOKEN (SCAN+1)='M' || MAT_INDX ;
```

```
            MAT_INDX=MAT_INDX+1;
            END;
        ELSE;
      GO TO LOOP;
   END PARSE;
```

各自，問 28〜31 に解答せよ．

32. データの型，長さ，精度，配列の場合にはその大きさ，構造の要素のとき接続情報．

実行プログラムは記憶域の割付けが要求されたときどのような変数が宣言されているかを知る必要がある．基底つき変数の場合も同様である．

33. 否．変数に対してはこれでよいが，プログラムの制御（回帰呼出しのレベル番号の記憶など）について何も処置されていない．このためにはスタック・フレームが必要である．（PL/I の RECURSIVE 属性を調べよ）．

34. a.　解釈段階は，行列の欄を作成する際に，記号表を検査することができる．最も簡単な方法は添字の値が最大値として指定された値を越えないようにすることである．

b.　単語段階という答えもあるだろう．（これはあまりよい方法ではないが……）長過ぎるとわかった記号は，N+M が記号の最大値として，最初のN文字と最後のM文字と最後のM文字とを結合して短くすることもできる．しかし，記号の正当性の検査は解釈段階で行なうのが最もよい．その理由は，記号が参照されているのかまたは多重に定義されているのかの区別がこの段階で行なわれるからである．単語段階ではこのような意味上の誤りやブロック構造に関連する誤りを検出することはできない．すなわち，次の第1の例におけるAは別物であり正しいと判定し，第2の例のAは間違っていると判定するような段階は単語段階の能力を越えている．

1.　DCL 1A STATIC,

```
            2B,
              3A CHAR(4);

   2. DCL 1A STATIC
            2A       CHAR(4),
            2A       CHAR(4),
```

c. 構文段階でこれを検出できる．IF トークンを発見すると一連の変形則を用いて変形し，THEN がないとわかったとき error ルーチンを呼出さずに，式の直後に THEN があるものとして処理することが可能である．

35. a. 〈DO 文〉::=DO ; | DO〈記号〉=〈式〉TO〈式〉BY〈式〉;

b. 変形則

8a ;　DO ; /do/′′**/7
8b ;　DO 〈記号〉/iterative/′′**/21
21 ;　=〈式〉/op_prec value/′′**/22
22 ;　TO 〈式〉/op_prec value/′′**/23
23 ;　BY 〈式〉/op_prec value/′′**/24
24 ;　;//′′**/7

活動ルーチン：

do——　DO 演算子を行列に入れる．
iterative—— DO 演算子を繰り返しの DO グループであるという表示をつけて行列に入れる．
value——　op_prec によって与えられた〈結果〉を行列に入れる．

c. 上のような方法を用いるためにはコード生成則と変形則を変更しなければならない．幾つかの活動ルーチンの追加と終端記号表に新しい欄 DO, TO, BY を追加すること以外は段階を変更する必要はない．

d. 上の変更に加えて．どの END 文がどの DO 文に対応するかを知るために，DO グループの入れ子の深さを記憶しておかねばならない．

e.

7a：　ADD〈任意〉/add/S1*/25

25：　〈記号〉TO/arg/''**/26

　　　〈記号〉, /arg/''**/25

26：　〈記号〉; /final_arg/''**/7

活動ルーチン：

add——　ADD 演算子を行列に入れる．

find_arg——最後の引数であることを表示する．これに他の引数の値が加算される．

36. INITIAL 属性を指定している宣言文に対しては，変数を初期化するための目的コードが生成される．

自動および被制御記憶域を定義した宣言文に対しては，実行時に適当な記憶域を割付けるための目的コードが生成される．

37. a. & b.　記憶域に関する問題を論じるには基底つき記憶域を考えるのが最もよい方法である．たとえば，文 A=B は，B が静的領域に置かれていれば，

A=STATIC→B

であると考えられる．

すなわち，すべての参照はその記憶域を指すポインタをあるレジスタにロードしてから行なわれると考えられる．

STATIC→は次のようになる．

L 15, STATIC

L AC, _(0, 15)

_は静的領域の中のBのオフセット

本問の文：

(P1→A . B . NEXT)→A . B . C

において，A . B . NEXT は記憶域を指すポインタであり，このポ

インタの記憶域を指すポインタ P1 が存在する．したがって：

(P1→A.B.NEXT) は次のようになる

L 15, P1

L 13, _(0,15)

_は配列Aの中の A.B.NEXT のオフセット

A.B.Cを指すポインタがレジスタ13に設定されるとすると：

(P1→A.B.NEXT)→A.B.Cは次のようになるL AC, _(0, 13)

_は構造体Aの中のCのオフセット

これまでの説明で理解されたように，問 a, b の答えは本文の説明でつくされている．

38. a. 実行時に記憶域を割付けるために，コンパイラはオペレーティング・システムを呼出す手続きを生成する．この呼出しはコンパイルされたプログラムを実行することによって行なわれる．どの記憶域が空いているかはコンパイル時にはわからない．記憶域の割付けは実行の度に異なることもある．プログラムの導入部で変数の番地表を管理することもできるが，記憶域割付けの問題はオペレーティング・システムの役割りである．

オペレーティング・システムの呼出しは：

CALL STORAGE_MANAGEMENT (POINTER, SIZE)

オペレーティング・システムはある大きさ SIZE の記憶域を割付けてその番地を POINTER に設定して返却する．

b. 条件とは，システムの標準的な処理によらず，プログラムの制御によって割込みを処理する手段である．このためには，割込みが発生したときユーザ・プログラムに制御を渡すように，プログラムの入口の番地を設定しておく必要がある．

また，オペレーティング・システムの標準的な処理とユーザ・プログ

ラムの処理との切換えを行なう機能も必要である．

シグナルとは人為的な割込み発生方法であり，オペレーティング・システムで簡単に行なうことができる．

c. すべての I/O は，読者が第9章で学ぶように，I/O チャネルと呼ばれ，独自の命令を持つ別個の計算機によって実行される．この計算機のプログラムを行なうのは難しい．I/O 計算機と CPU との通信は割込みによって行なわれるが，割込み処理ルーチンを作成することはもっと難しい．

コンパイラが I/O チャネル・プログラムを生成するのは困難である．たとえコンパイラが I/O チャネル・プログラムを生成するにしても，割込み処理の用意をもしなければならないのは負担が大き過ぎる．

このような場合にはオペレーティング・システムのサブルーチンを用いる方がよい．コンパイラは単に適当なオペレーティング・システムのサブルーチンへの呼出し手順を生成し，すべての I/O 動作をオペレーティング・システムに依頼する．

第9章　オペレーティング・システム

1. a. チャネルは基本的には特殊目的のプロセッサであり，CPU に従属して I/O を実行するために使用される．

b. プログラム実行を，I/O 完了を待合せているとき以外は，I/O 実行と非同期に行なうことができ，データ転送の度に CPU に対する割込みを発生させる必要がなく，システムの仕事量が増加させることができる．

c. チャネル状態語はチャネルの動作に関連するすべてのデータを納めている．これはちょうどプログラム状態語（PSW）が CPU で実行中のプログラムのすべてのデータを納めているのと同じである．

CPU は CSW を調べて動作の進行状況を検査することができる.

d. チャネルは I/O 動作の完了時に CPU に割込むことができる. データ転送が正常に行なわれているのであればプログラムはI/O完了を待合せればよい. 異状があればオペレーティング・システムはオペレータを呼出すなどの処置をとらなければならない.

2. CPU が本問のような原始的な I/O 機能しか持たないときには次のことを行なわねばならない：

a) 装置の動作モードを適当に設定するために“設定コマンド”を出力する.

b) 装置の動作モードが設定されるのを待つ.

c) 文字を読込んだり書出したりする.

d) (b)に戻り待合せる.

平均の命令実行時間が10マイクロ秒（やや遅い計算機である）としても CPU は 99.99% までアイドルとなる.
割込みの機能を利用すると, 装置が空いたという報告が行なわれるまで, CPU は（他の）プログラムを実行したりして無駄をなくすることができる.

3.

CCWの番地	**機　能**
1600	次のページの先頭まで紙を送る.
1608	“***** *****” と印刷して1行送る.
1610	“***** *****” と印刷して1行送る.
1618/1620	“JOB NUMBER 6.251” と印刷して1行送る.
1628	“**********” と印刷する.
1630	“**********” と印刷して次のページの 先頭まで送る.

4. 38番地：I/O “旧” PSW

割込みが起こって I/O ルーチンに制御が移されるとき，CPU は現在の PSW をここに格納し，I/O ルーチンが完了したとき現在の PSW として復元する.

40番地：チャネル状態語（CSW）

チャネルは，I/O 動作の終了時に動作の状態を示すデータをここに設定する（正常終了，異状終了など）

48番地：チャネル番地語（CAW）

ユーザ・プログラムはI/O要求に先立ってここにチャネル・プログラムの番地を設定している.

78番地：I/O "新" PSW

ユーザは，I/O 割込み処理ルーチンの番地をここに設定しておく. CPU は，割込みが起こったとき，この内容を現在の PSW として使用する.

5. 記号による I/O "アセンブラ語" を作成することは可能である. 命令記号と記号番地は便利である. このアセンブラ語の処理は通常のアセンブラと同様である.

b.

READ, WRITE, SKIP, READ CARD READER (RCR), TEST, ……など.

c. 根本的な変更は不要である. もちろんデータ・ベースの内容は異なっている. たとえば機械命令表で |L|58| などはなく，|WRNOSPACE*|01| などがある.

d. たいていのI/Oルーチンは非常によく似ている. システム・マクロを使用することによってプログラミングの負担が軽減される.

6. IBM 360 を IBM 370 と同じように動作させるようにする方針で解答す

* 印刷機に対するコマンドの1つである.

る.

IBM 370 では語を参照するとき語境界に整列していなくてもよく, A 1, 1317 という命令を実行してもプログラム割込みは生じない. (しかし, 語境界に置かれていない語を参照すると, 語境界に置かれている語を参照するのに比べて, 10倍も時間がかかる.)

ここでは A 1, 1317 という命令が出てきたとき 1317 番地から4バイトを適当な語境界の領域に置き直し, 新しい命令 A 1, DATAを作って実行することにする. この命令を実行した後に制御をユーザ・プログラムに返却する.

この問題を解決するためにはユーザ・プログラムを変更せずに, 次のような割込み処理プログラムを作る:

簡単のためにこのルーチンは 0〜4095 番地に格納する.
このようにするとベース・レジスタを使用しなくてもよい.




```
        STM    1,2,SAVE              R1およびR2を退避する
        L      1,44                  PSWから番地を求める
        LA     1,0(0,1)              第1バイトを消去する
        S      1,=F'4'               割込みを起こした命令の番地を求める
        MVC    INST1(4),0(1)         その命令を複写する
        MVC    INST2(2),0(1)         その命令を複写する
        XC     INST1(2),=X'000F'     命令コードとR1を消去する
        OC     INST1(2),=X'4120'     LA 2,... に変更する
        EX     0,INST1               R2に実効番地を計算する
        MVC    TEMP(4),0(2)          語境界にデータを移動する
        XC     INST2+1(1),=F'F0'     INST2からX2を消去する
        EX     0,INST2               命令を実行する
        LM     1,2,SAVE              R1およびR2を復元する
        LPSW   40                    次の命令から続行する

INST1   LA     2,0(0,0)              命令の見本 (ここに命令が作られる)
INST2   A      0,TEMP                命令の見本 (ここに命令が作られる)
TEMP    DS     F                     語境界に設けたデータ領域
```

このハンドラは RX 型命令（たとえば A, S, M, L など）のインデックスおよびベース・レジスタを含んだ番地指定の割込みを処理する．

註：上述のプログラムは次の欠点を有している：

1）番地指定でないプログラム誤により割込みが生じてもこのルーチンが実行される．

2）このプログラムは格納命令を処理しない．

7. b.　bおよびdは純粋で修飾されない．

記号表，コードおよび行列は純粋でない．アセンブラやコンパイラが多重プログラムの環境で使用されているとしても，純粋でないこれらのデータ・ベースを共有することはできないので，同時に幾つかの原始プログラムを処理することはできない．

8. 断片は，多重プログラムの環境で連続していない記憶域が開放されるときに生じる．各記憶域は1つだけでは待ち行列の中のプログラムを納めるのには小さすぎるが，全体の容量の和は必要量よりも大きいことがある．これを解決する方法は再配置であり，プログラムを移動させて空きを作ったり，使用できる領域に入るようにプログラムを分割したり（ページング）する．

9. a.

ページ	**セグメント**
名前を持たない．	名前を持つ．
物理的な単位．	論理的な単位．
固定長（幾とおりかの長さがあることがある．しかし，任意長ではない）．	可変長（ユーザの要求によって定められる）．
ユーザにはわからない．	ユーザにわかる．

b.　可能．　ページングなしのセグメンテーションおよびセグメンテーションなしのページングはいずれも可能である．

10. 純手続きは自分自身を変更しない手続きである．すなわち，手続きの中の

データや命令で変更されるものはない．

２人以上の利用者が１つの手続きを同時に用いようとするとき，およびリエントラントな手続きが必要とされるとき純手続きが使用される．
純手続きの第３の利点は読出し専用であることである．読出し専用のセグメントに対しては，ページングでオーバレイされるページを２次記憶に書き出さなくてもよい．

利点：1）　共用が可能
　　　　2）　回帰呼出しが可能
　　　　3）　読出し専用

11. 本問は極限状態に関する質問である．
CPU が平均的に３つのプログラムしか同時に処理できないとしたら，プログラムをコアに入れるためにオーバヘッドの大きい複雑な記憶管理技法を用いると却って全体の仕事量を減少させることになる．基本的には，システムが記憶域制限のときは記憶管理を行なうとよい．CPU 制限のときや I/O 制限のときは記憶管理は役に立たない．たいていのシステムは極限状態ではないが，それらの間の状態をとる．

12. 問に示した順で番地空間割付けの柔軟性と複雑さやオーバヘッドが増加する．

a.　単一連続割付け
利点：容量の大きな記憶域が使用できる．
欠点：ユーザを切換えるオーバヘッド．
　　　　断片．

b.　再配置型分割割付け
利点：(a)の利点に加えて，断片の問題が解決されること．
欠点：分割領域の再配置のオーバヘッド．

c.　単純なページング
利点：断片の問題が能率よく解決できる．

欠点：コア容量よりセグメント長を大きくすることはできない．
ページングを行なうためのハードウェアが必要．

d.　デマンド・ページングを行なうセグメンテーション

利点：記憶管理の観点からは，セグメンテーションを行なうとダイナミック・リンキングが可能となる．
ダイナミック・リンキングを行なうと，セグメントが実際に参照されるまでは記憶装置と CPU 時間を使用しない．さらにセグメンテーションは手続きの共用を可能とし，記憶領域の節約となる．
ページングは断片を解決する．

欠点：セグメンテーションを行なうためのハードウェアが必要．

13. a.　略式にいうと，番地空間とはユーザが処理している記憶域である．システムのハードウェアの装置は番地空間を物理的な記憶域に（あるいは逆に）種々の複雑さで写像する．

b.　1）　番地空間に定数を加えて写像する．

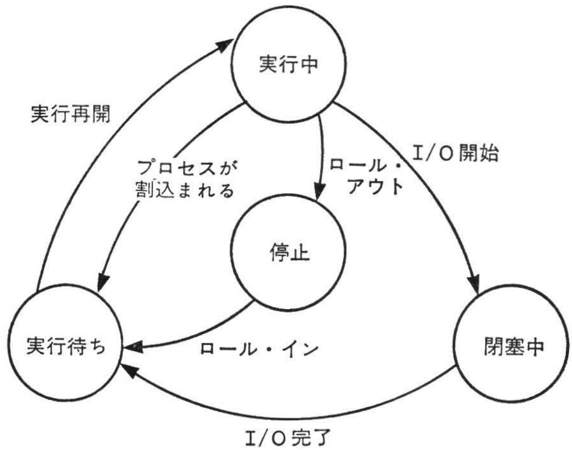
2）　詰め合せを行なって大きな空き領域を作るために，プログラムが再配置されるとき定数が変更されること以外は，(1)に同じ．

3）　番地空間は２次元的に表現される．番地空間には複数のセグメントが存在し，各セグメントは別個の一次元の番地空間を構成する．

ページングは各セグメントを一定の長さの単位に分割し，各単位を，ページごとに異なった定数を加えて，コアのブロックに写像する（コアのブロックは連続している必要はない）．

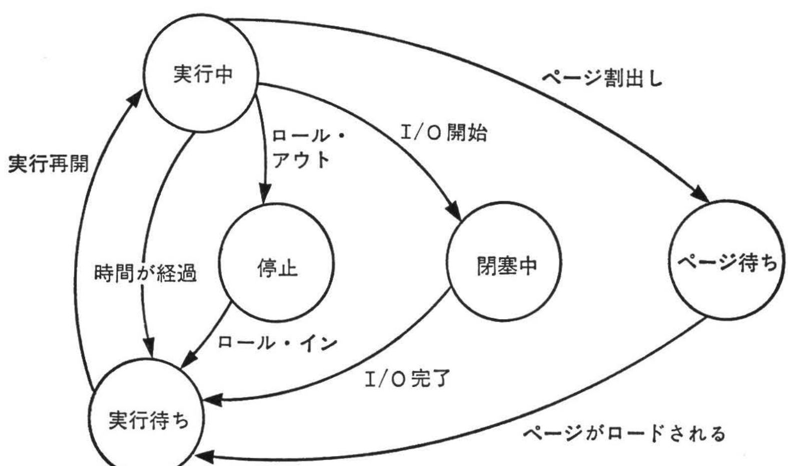
14.　１つのプロセスに関連したことはすべて記憶装置（および外部記憶装置）に納められている．CPU は単に記憶装置の内容にしたがって動作する．したがって，どの CPU が割当てられようと全く同じことである．

15. a.

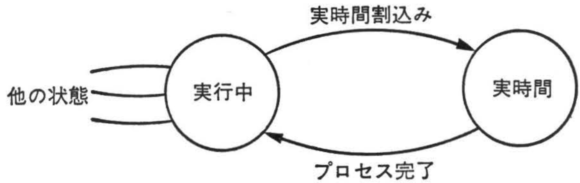


b.　閉塞中の状態を分割する．

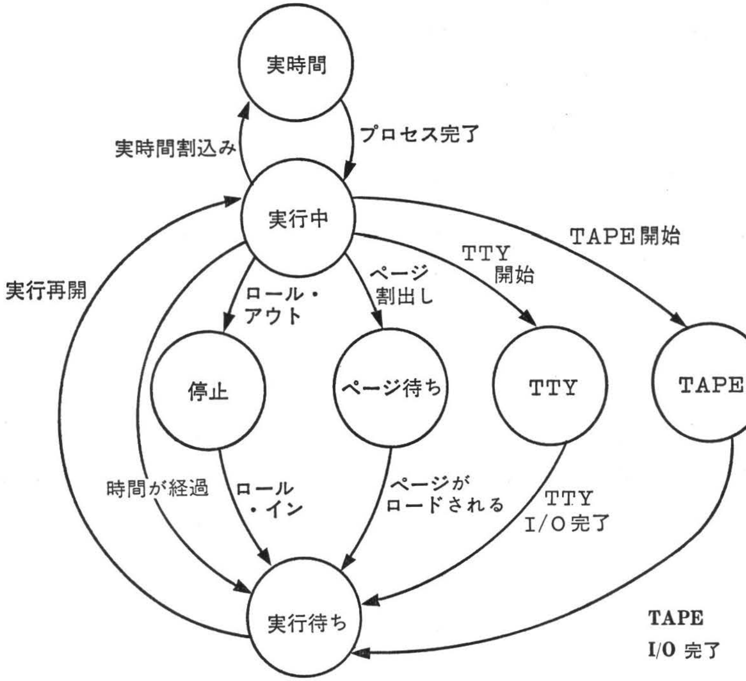
利点：ページ待ち状態から実行待ち状態となったプロセスは実行待ち行列の先頭に置くことが望ましい．



c.　次のループを追加する：



d.　TTY によるデータ転送は TAPE よりも数桁低速である．閉塞状態を TTY 状態と TAPE 状態に分割する．



16. 競走は2つ以上のプロセスが同一の装置を“同時”に要求するとき起こる．（第9.3.3節参照）

競走は，装置を“要求”し，処理が終了したときに“解放”することによって解決できる．

手づまりは，2つのプログラムが互に他が必要とする資源を解放せずに，すくみ合って閉塞状態となる競争をいう．

現在のところよい解決方法はない．

17. 多重処理システムにおいては，2つ以上のプロセッサが同時に同一のプロセスの実行を開始しないようにするために，プロセッサ間の通信方法の1つとして，ソフトウェア・プロセッサ・ロックアウトを行なう。

他の例はI/O チャンネル（I/O プロセッサ）とCPU 間で見られる．CPU はI/O チャネルで印刷すべき情報をバッファ領域に入れる．チャネルがバッファの内容の印刷を終了したときCPU に完了を報告すると，CPU は再びバッファに書込むことができる．このようにして能率のよい動作を行なうことができる．

18. **スプーリング**（SPOOLING：オンライン同時周辺動作）は，印刷機などの低速の装置が使用できるまで，テープやディスク駆動装置などに入出力情報を格納しておく方法である．

緊急の場合やプロセス制御などの実時間動作の場合には直ちにI/O 動作を行なう必要があり，スプーリングを行なうことは適当でない．

19. I/O トラフィック制御は，1つのI/O 装置が一時に1つのプロセスによって使用されるようにし，幾つかのプロセスからの要求が生じたとき，その装置が使用できなければ，装置に対する待ち行列を作って管理する．

20. 順アクセス記憶装置を使用するのに適した応用例は，分類した形式で電話番号簿を印刷する場合である．

直接アクセス記憶装置を使用するのに適した応用例は，計算機化した電話による情報サービスの（たとえば，Stuart Madnick の電話番号は何番かなどといった）場合である．

21. OPEN は，（頻繁に使用されるであろう）ファイルに関する情報をコアに複写する．この情報はファイルの書式や格納場所を含む．もしファイルが

頻繁に使用されるならば，ファイルに関する情報を調べる度に2次記憶をアクセスしなくてもよいようにするべきである．

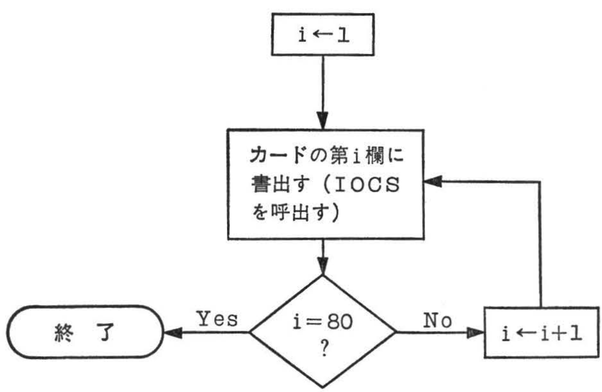
CLOSE は不要となった情報をコアから取り除く．

22. セグメンテーションはファイル・システムに含められるべき機構を持っている．たとえば，ファイルを追加したり，ファイルの内容が削除されたときファイルを詰め合せたりすることは簡単に行なえる．プログラムやデータの保護とアクセスの制御はセグメンテーションの機構によって達成される．

23. a.　可能．この単純な計算機は物理的なI/O機構にのみ関係するが，論理的なファイル編成には影響しない．

b.　入出力制御システム（IOCS）および装置駆動モジュール（DSM）に影響がある．他のモジュールには関係がない．

c.　テープに作られる順ファイルであると仮定する．DSM ルーチンは次のようになる．



24. a.　連続再使用可能なプログラムは，1回の実行が終了した後でのみ再実行できる．この種のプログラムは実行開始時に掃除，すなわち初期化を行なう．プログラムを使用しているプロセスが終了するまで他のプロセスはこのプログラムの実行を開始できない．純手続きは自分自身を変更することがないので，任意の時点に任意個数のプロセスが実行

を開始したり終了したりすることができる．純手続きは多重処理や共有の目的には都合がよい．

b. 否．タイム・シェアリングのためにはページングやセグメンテーションは必ずしも必要でない．これらの技法を用いるとシステムの動作がより効率よくなり，利用者にはより多くの柔軟な機能が使用できる．

c. 各セグメントは必要になったとき割付けられる．

d. 各セグメントは個別に定められた条件にしたがって保護することができる．

25. a. (1) 主記憶：コア

(2) 2次記憶：ドラム，ディスク

b. (1) セグメンテーション：

ファイル・システム
コア
DBR
PBR
再配置レジスタ
LP
AP
SP

(2) ページング：

ディスク
ドラム
ファイル・システム
コア
テープ
DBR
再配置レジスタ

c.　ダイナミック・リンキング：

ファイル・システム

ジョブ・スケジューラ

コア

DBR

LP

AP

SP

再配置レジスタ

PBR

d.　ローディング：

ファイル・システム

PBR

DBR

再配置レジスタ

e.　1）　ファイル・システム，ジョブ・スケジューラ，コア

2）　ファイル・システム，ジョブ・スケジューラ，コア，ファイル記述子のファイル

3）　ファイル・システム，ジョブ・スケジューラ，コア，ファイル記述子のファイル

4）　ファイル・システム，ジョブ・スケジューラ，コア

5）　ジョブ・スケジューラ，ドラム，ディスク，コア，拡張記憶装置

6）　(5)とファイル・システム

7）　(5)に同じ

26. 結合用のデータ（リンク）は手続きを純粋でなくする．

27. リンケージ・セグメントの番地はリンケージ・ポインタ・レジスタ（LP）に納められている．1つのセグメントを使用する前にはLPにリンケージ・セグメントの値を設定しておかねばならない．この方法の1つとしては

リンケージ・セグメント自身によって適当な値を設定する方法がある．

28. a.

READ FILE (ALPHA) RECORD(4) SIZE(55) LOCATION(BUFF);

1) VTOC の最初の部分を調べるためにブロック0を読込む．
2) VTOC を探して ALPHA に対する欄を求める．このファイルのファイル・マップはブロック4にあることがわかる．
3) 論理バイト位置を計算する [(4−1)×55=0165].
4) 論理バイト位置0165は論理ブロック0のバイト165に対応する．
5) ALPHA のファイル・マップ（ブロック4）をバッファに読込む．
6) 論理ブロック0に対する物理ブロックを求める（ブロック3）．
7) ブロック3をバッファに読込む．
8) バイト165—219をバッファから取り出し，BUFF に移す．

b. READ FILE (ALPHA) RECORD(19) SIZE(55) LOCATION(BUFF);

1) VTOC の最初の部分を調べるためにブロックを読込む．
2) VTOC を探して ALPHA に対する欄を求める．このファイルのファイル・マップはブロック4にあることがわかる．
3) 論理バイト位置を計算する [(19—1)×55=0990].
4) 論理バイト位置 0990 は論理ブロックのバイト990に対応する．
5) ALPHA のファイル・マップ（ブロック4）をバッファに読込む．
6) 論理ブロック0に対する物理ブロックを求める（ブロック3）．
7) ブロック3をバッファに読込む．
8) バイト990—999を取り出し，BUFF から BUFF+9 に移す．これは要求された55バイトの内の始めの10バイトであり，次の45バイトを読込む必要がある．
9) ALPHA のファイル・マップ（ブロック4）をバッファに読込む．
10) 論理ブロック1に対する物理ブロックを求める（ブロック7）．
11) ブロック7をバッファに読込む．
12) バイト 0—44 を BUFF+10 から BUFF+44 に移す．

29. a.　IBM 360 のベース・レジスタは純手続きを組込むのに使用できる．またコアはブロックごとの保護機構を備えている．

H645 は間接番地機能，セグメンテーションおよびページングのためのハードウェア，および純手続きを組込むためのレジスタ（AP，LP，SP）を備えている．

b.　セグメンテーションは保護のためによい方法である．セグメンテーションを用いるとセグメントごとに別個に定めた条件により保護が行なえる論理単位（コアの定長のブロックによる保護ではないことに注意）を構成することができる．

すべての番地指定はセグメント表を通じて行なわれるので，ハードウェアにより，すべての参照に対して正当性の検査を行なうことができる．この検査を行なうためにセグメント表の中の情報が使用される．

30. a.　MULTICS においては，セグメントは参照されたときにロードされ，その直後にセグメントを参照するために必要な結合が行なわれる．しかし，ロードされたばかりのセグメントにより参照される他のセグメントはこれらが実際に参照されるまで結合されない．

b.　再配置のための RLD のカードは必要でないが，結合情報を与えるための RLD カードは必要である．

c.　記述子はセグメントの格納場所，長さ，型および保護のクラスを納めている．

31. a.　ローダ（リンカ）．

d.　JOHN のリンケージ・セグメント．

c.　JOHN およびそのリンケージ・セグメントを生成したコンパイラ．

d.　SQRT のリンケージ・セグメントは LP に適当な値を設定しなければならない．(註)

註：本文中で説明したように，LP を設定するには他の方法も可能である．実際に，最近では別の方法に変更されている．

32. “一組のレジスタを用いる” 方法は，セグメンテーションを行なうのには適当でない．この方法ではレジスタの個数でセグメントの個数が制限されてしまう．

CPU のロケーション・カウンタはセグメント番号と語番号(たとえば<32, 177>）を納めている．
SWAP_DBR がすべての番地空間において同じセグメント番号を持っていないとしたら，（例では32）新しい番地空間で別の場所に対応してしまう．

33. システムが直面している制限因子に関係する対応策のみが有効である．解答は読者にまかせる．

34. よい勉強になる演習である．やってみるがよい．

索　　引

A

B

K

L

M

R

S

MEMO

MEMO

M E M O

MEMO

MEMO

M E M O

≪訳者略歴≫

池田克夫
いけだかつお

1937年　滋賀県に生まれる.
1960年　京都大学工学部電子工学科卒業
1965年　京都大学大学院工学研究科博士課程終了
1965年　京都大学助手
1971年　京都大学工学部情報工学科助教授
文部省在外研究員としてユタ大学および
マサチューセッツ工科大学に留学.

現　在：筑波大学電子・情報工学系教授

システム・プログラム　II

昭和51年10月15日　初版発行
昭和53年5月15日　2刷発行
昭和54年10月15日　3刷発行
昭和56年4月15日　4刷発行
昭和57年4月15日　5刷発行

著　者　J. J. ドノバン
訳　者　池田克夫
発行者　村越勝弘
発行所　**日本コンピュータ協会**
東京都文京区本郷3丁目5―5
電話　東京（03）813―6929